2021

Perils and pitfalls of symbolic regression

Ryan Grindle
University of Vermont

Perils and pitfalls of symbolic regression

A Thesis Presented

by

Ryan Grindle

to

The Faculty of the Graduate College

of

The University of Vermont

In Partial Fulfillment of the Requirements
for the Degree of Master of Science
Specializing in Computer Science

August, 2021

Defense Date: July 14, 2021
Thesis Examination Committee:

James P. Bagrow, Ph.D., Advisor
Mads Almassalkhi, Ph.D., Chairperson
Josh Bongard, Ph.D.
Cynthia J. Forehand, Ph.D., Dean of the Graduate College

Abstract

The ever-growing accumulation of data makes automated distillation of understandable models from that data ever-more desirable. Deriving equations directly from data using symbolic regression, as performed by genetic programming, continues its appeal due to its algorithmic simplicity and lack of assumptions about equation form. However, few models besides a sequence-to-sequence approach to symbolic regression, introduced in 2020 that we call y2eq, have been shown capable of transfer learning: the ability to rapidly distill equations successfully on new data from a previously unseen domain, due to experience performing this distillation on other domains. In order to improve this model, it is necessary to understand the key challenges associated with it. We have identified three important challenges: corpus, coefficient, and cost. The challenge of devising a training corpus stems from the hierarchical nature of the data since the corpus should not be considered as a collection of equations but rather as a collection of functional forms and instances of those functional forms. The challenge of choosing appropriate coefficients for functional forms compounds the corpus challenge and presents further challenges during evaluation of trained models due to the potential for similarity between instances of different functional forms. The challenge with cost functions (used to train the model) is mainly the choice between numeric cost (compares y-values) and symbolic cost (compares written functional forms). In this work, we provide evidence for the existence of the corpus, coefficient, and cost challenges; we explore why these challenges exist in the model, and we propose possible solutions. We hope that this work can be used to initiate improvements to this already promising symbolic regression model.

Acknowledgements

Thank you Jim Bagrow and Josh Bongard for your guidance, thoughtful discussions, and everything you have taught me. It has been a pleasure to work together. Thank you members of the Morphology Evolution and Cognition Laboratory and Lapo Frati for sharing you interesting research with me and allowing me to share mine with you. Thank you friends and family for your love and support. Thank you to Mads Almassalkhi for being the chair of my defense committee. Thank you to the University of Vermont and CA technologies who funded me as a Graduate Research Assistant.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Faced with a growing flood of data, researchers are increasingly motivated and challenged to understand the underlying patterns and relationships hidden in raw data. Equations that immediately predict current and future data generated by some domain are appealing, as mathematics is dedicated to expressing complex concepts in brief, readable notation. But, before an equation can be used to aid in understanding data, the equation itself must be manually constructed or automatically discovered. For most problems, manual construction is infeasible, so automated methods are increasingly preferred.

Regression is a powerful tool for finding equations. However, it is not always obvious which functional form to choose, against which coefficients should be fit. Symbolic regression on the other hand does not presuppose a functional form and is usually performed with genetic programming (GP) [17]. When applied to symbolic regression, genetic programming trains increasingly accurate candidate equations of differing forms, and thus allows for little to no assumptions about the functional form of the final equation. However, GP is normally performed on data from just one domain at a time. Thus, it is poor at transfer learning: it rarely predicts new data generated by an unseen

domain without significant additional training.

Some neural networks have been shown capable of transfer learning (e.g. [21, 32]) by learning relationships that transcend any one domain. But, those networks were trained for tasks other than symbolic regression.

Other neural networks have been used to perform symbolic regression such as [7, 30, 15, 28, 1], but none of these neural network exhibit transfer learning. The neural network in [7, 30, 15] use their weights to describe the underlying equation of a single regression dataset and therefore do not exhibit transfer learning. The neural network in [1] is trained to generate equations without taking data as input. Instead, the dataset was used to compute fitness or reward, which was then used to update the weights of the neural network. Thus, those networks cannot generate new equations from data generated by a novel domain without further training, and are therefore incapable of transfer learning. In [28], the neural network behaves in much the same way that genetic programming behaves. In each generation, the neural network generates a population of equations/trees and its weights are updated based on the error of those equations against the target data. So, just like genetic programming, this neural network requires additional training to produce a low error equation on a domain on which it has not been trained. Another method reports neural networks capable of symbolic regression [13], but it is not clear whether the resulting networks are capable of transfer learning.

1.1 Transfer learning and symbolic regression

Work by [20] introduces a method capable of training networks capable of both symbolic regression and transfer learning, but the method requires prior knowledge, in the form of asymptotic constraints, about the domains to be modeled. Like determining functional forms, determining asymptotic constraints takes time. This is the time that we are trying save by using symbolic regression (instead of regression) in the first place.

Prior work has incorporated a form of transfer learning into GP [12, 8, 25, 27, 4]; however, evolution was performed on both training and testing domains. All these approaches used a training domain to learn valuable trees or subtrees, called building blocks, which were then used during training on the test domain. The authors of [8, 12, 27] included these building blocks in the initial population for the test domain. The authors of [25] used the building blocks as elements of the primitive set when training on the test domain. [4] used performance on the test domain to alter the weight of influence of different instances of the training data. Thus, in all these methods, the learner's structure was altered by GP using feedback from the test domain.

Symbolic integration and solving of differential equations was achieved using a sequence-to-sequence model in [18], and [19], which demonstrated transfer learning in their respective domains; however, neither approach was applied to symbolic regression.

In the method proposed by [3] (also see Section 2.4) trained learners can predict data from a test domain without requiring further alteration to their

internal structure; thus exhibiting transfer learning. Recent work in [3], introduced a convolutional sequence-to-sequence neural network [10] capable of transfer learning in symbolic regression. Their model takes normalized y-values located at fixed x-values ($x = \{0.1, 0.2, \cdots, 3.0\}$) and outputs the functional form of the underlying equation. The coefficients of the functional form are then determined with a nonlinear optimization algorithm. Their model is capable of transfer learning because without further alteration to the weights after training, the neural network performs symbolic regression on functional forms that it has never seen and functional forms that it has seen with different coefficients. The authors show that the functional forms produced by the neural network are able to generalize better to an extrapolation region than a numeric regression neural network.

Figure 1.1 illustrates the difference between typical symbolic regression algorithms like GP (A) and symbolic regression algorithms capable of transfer learning (B) like the work in [3].

In this work, we will continue to discuss the model in [3] (which we call y2eq). We identify three difficult challenges that any model capable of transfer learning in the task of symbolic regression will face. The challenges are the corpus, coefficient, and cost challenges. The challenge of devising a training corpus stems from the hierarchical nature of the data since the corpus should not be considered as a collection of equations but rather as a collection of functional forms and instances of those functional forms. The challenge of choosing appropriate coefficients for functional forms compounds the corpus challenge and presents further challenges during evaluation of trained models due to the potential for similarity between instances of different functional forms. The challenge with cost functions (used to train the model) is mainly

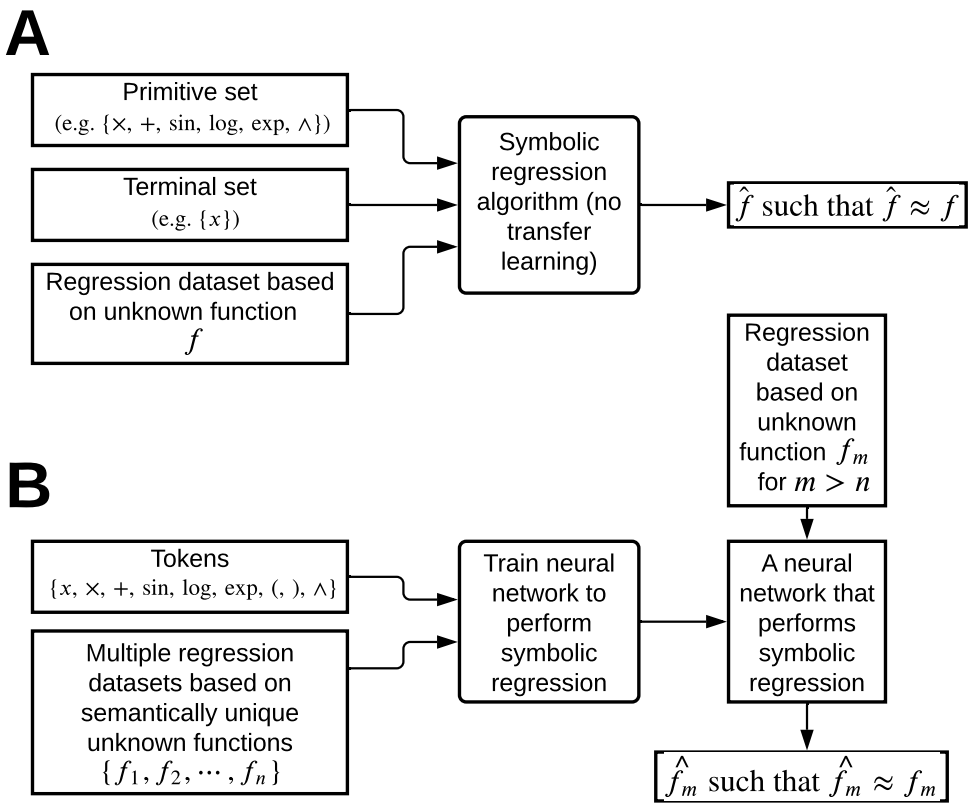

Figure 1.1: This figure compares a typical symbolic regression algorithms (A) and the training of a neural network capable of transfer learning in symbolic regression (B).

the choice between numeric cost and symbolic cost. These challenges are described in more detail in Section 3.

Chapter 2

Background

This section provides background information on many of the important topics covered in this thesis.

2.1 Artificial neural networks

Artificial neural networks, which will be shortened to neural networks in the remainder of this thesis, are mathematical functions that have parameters called weights. For example, a neural network with parameters w that accepts inputs x can be denoted as $N_w(x)$.

Neural networks are typically described as directed graphs. Along each link is one of the weights. Each node holds a single real number. The inputs to the neural network travel through the graph and get multiplied by every weight and added to other incoming values at each node. Additionally, activation functions are applied at nodes after the summation. Examples of activation functions include: tanh, sigmoid, softmax, ReLU, step function. Figure 2.1 shows a simple example of a neural network.

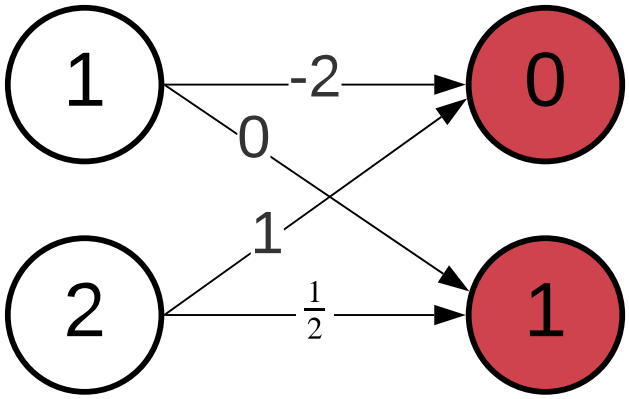

Figure 2.1: Here is an example of a artificial neural network where the activation function is the identity function I. The inputs are 1 and 2 and the outputs are 0 and 1. In other words, $N_w(\{1,2\}) = \{0,1\}$). Using the weights drawn on each link, the exact equations used to compute the values at the output nodes (red) are seen to be $0 = I(1 \cdot -2 + 2 \cdot 1)$ and $1 = I(1 \cdot 0 + 2 \cdot \frac{1}{2})$.

The numbers that reach the final layer in the neural network (called the output layer) can be compared with target values using a cost function that measures the error in the predicted output versus the target output. Since all operations in neural networks are purposely chosen to be differentiable, the error associated with each output can be translated into an adjustment to the set of weights through a process called backpropagation. There are other ways to train neural networks, but backpropogation is the most common. Just a few of the tasks that neural networks can be trained on are regression, classification, language translation, image captioning, and robot control.

2.1.1 Sequence-to-sequence neural networks

A sequence-to-sequence (seq2seq) neural network receives an input sequence and produces an output sequence. A typical use for such a model

is translations (i.e. English to French). Sequence-to-sequence models are composed of two main components: an encoder and a decoder. The encoder reads the input sequence and produces a context vector which is passed to the decoder. The decoder also receives the previously output elements of the output sequence and uses this in combination with the context vector to produce the desired output. Both the encoder and the decoder are recurrent neural networks, which allows them to handle sequences.

We will use the notation from [5], which contains a clear explanation of seq2seq neural networks. Let the input and target output and predicted output sequences of a seq2seq model be defined as $X = \{\vec{x}_1, \vec{x}_2, \cdots, \vec{x}_n\}$ and $Y = \{\vec{y}_1, \vec{y}_2, \cdots, \vec{y}_m\}$ and $\hat{Y} = \{\hat{\vec{y}}_1, \hat{\vec{y}}_2, \cdots, \hat{\vec{y}}_m\}$, respectively.

Let the encoder be represented as $\vec{h}_t = f(\vec{x}_t, \vec{h}_{t-1})$ where $\vec{h}_t$ is the hidden state of the recurrent neural network (RNN) at timestep t and f applies the weights of the RNN to the input and the hidden state. Notice that, as the name suggests, the recurrent neural network receives its previous value as part of its input. The encoder creates a vector $\vec{c}$ (called the context vector), which often is $\vec{h}_n$ (the final hidden state).

Let the decoder be represented as $\hat{\vec{y}}_t = g(\hat{\vec{y}}_{t-1}, \vec{s}_t, \vec{c})$ where $\vec{s}_t$ is the hidden state of the decoder. Like the encoder, the decoder is also recurrent, so we can see that the decoder receives its previous output state as input. Typically, $\vec{c}$ will be used as the initial hidden state of the decoder, namely $\vec{s}_0$.

During training of a seq2seq model teacher forcing is used, which means that the previous output of the decoder is forced to be the target output. This results in a slight alteration to the above decoder equation resulting in $\hat{\vec{y}}_t = g(\vec{y}_{t-1}, \vec{s}_t, \vec{c})$.

The model will be evaluated by categorical cross entropy (Equation 3.1),

which measures the correctness of each predicted token $\hat{\vec{y}}_t$ compared with the target token $\vec{y}_t$. This measurement is then used to apply backpropagation, which should improve the performance of the model.

At test time, the seq2seq model will have teacher forcing removed, so that it can be exposed to inputs for which the correct output is not know ahead of time.

2.1.2 Attention

Attention is a mechanism, used in neural networks, for focusing on important parts of the input sequence while determining the next element of the sequence. Sequence-to-sequence models with attention virtually always outperform seq2seq models without attention.

With attention there are many context vectors: one for each $\vec{y}_t$. These context vectors are computed as a weighted average on the hidden states of the encoder. The weights are computed based on the "alignment" of the hidden state of the encoder and of the decoder. In other words the context vector $\vec{c}_t$ that corresponds to output $\vec{y}_t$ is given by

$$\vec{c}_t = \sum_{j=1}^{n} \alpha_{tj} \vec{h}_j \tag{2.1}$$

where

$$\alpha_{tj} = \frac{\exp a(\vec{s}_{t-1}, \vec{h}_j)}{\sum_{k=1}^{n} \exp a(\vec{s}_{t-1}, \vec{h}_k)} \tag{2.2}$$

and n is the length of the input sequence and a is a function that measures the alignment of the two hidden states. There are many definitions that can be applied to a including letting a be a neural network. Notice that the expo-

nential functions used here mean that $\sum_{j=1}^{n} \alpha_{tj} = 1$. This use of exponential functions is often called the softmax function.

Once the context vectors are calculated, they are input to the decoder on every iteration, but this time the context vector changes from one iteration to the next. This type of attention is call soft attention [2].

There are many different variations of attention. For example, the convolutional seq2seq neural network in [10] uses a slight modification to the attention just described. Namely, residual connections are used when computing the context vector as can be seen in Equation 2.3.

$$\vec{c}_t = \sum_{j=1}^{n} \alpha_{tj}(\vec{h}_j + \vec{x}_j). \tag{2.3}$$

The difference between Equation 2.3 and Equation 2.1 is the addition of $\vec{x}_j$. Essentially the originally input vector $\vec{x}_j$, which was was used to compute the hidden state of the encoder $\vec{h}_j$, is being added to the vector it helped create $\vec{h}_j$. This puts a focus on the original input and allows the hidden state of the encoder to focus on a creating an effective combination of all the inputs.

Scaled dot-product attention

Scaled dot-product attention is used in transformers [33]. Transformers are also said to use self-attention because often the inputs to the attention mechanism all come from the previous layer. This is different to the types of attention discussed so far because those types of attention always connected the encoder and the decoder.

Scaled dot-product attention is defined as

$$A_d(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.4}$$

where Q is the query, K is the keys, V is the values, d_k is the dimension of the keys and the query. In Equation 2.4, softmax $\left(\frac{QK^T}{\sqrt{d_k}}\right)$ are weights that are applied to V.

Scaled dot-product attention is used in the following way: Q is a single element in the sequence and K is the entire input sequence. Thus, the single element of Q is dotted with the other elements in the sequence to determine weights for V. Usually $K = V$. The scale term $1/\sqrt{d_k}$ helps to ensure that the attention is not so strong that it removes all other information from the vector.

To make this mechanism trainable, weights are added to each component. That is, in practice weights W_Q, W_K, and W_V are applied so that the attention looks like $A_d(W_Q Q, W_K K, W_V V)$. Due to the success of transformers [33], scaled dot-product attention has become extremely popular.

2.1.3 Convolutional sequence-to-sequence neural networks

The paper [10], introduced convolutional seq2seq neural networks. These neural networks use convolution and (at least during training) avoid using any recurrent components which results in a big performance boost. The nature of convolutional neural networks also provides a natural way for neural networks to focus on adjacent tokens at the same time. This model also uses multi-

ple attention layers and then averages them together. The intuition is that an ensemble of attention mechanisms is better than an individual attention mechanism.

Because sequences are input all at once in the convolutional seq2seq architecture rather than sequentially, the neural network needs to be informed of the intended order of the inputs. This is done using a positional encoding. There are equations that have been used to compute positional encodings based on the token numbers such as that in [33]. Another popular approach is the use of an embedding layer, which maps token number to a trainable positional encoding. In other words, embedding layers can be used to learn positional encodings.

Positional encodings are simply added to input vectors to give the neural network an understanding of the intended order of the input. Convolutional seq2seq neural networks use a variation on soft attention described in Equation 2.3.

2.1.4 Transformers

Transformers [33] use scaled dot-product attention and self-attention. Like the convolutional seq2seq model, transformers also avoid recurrent components, which results in faster computation. Further, the matrix format for the attention mechanism allows for large amounts of parallelism and vectorization. Self-attention allows the transformer to compare every token in the input sequence to every other token in the input sequence. This is import for understanding the input sequence regardless of the domain. Transformers are a state-of-the-art architecture for sequence-to-sequence problems.

2.2 Regression

The process of regression is taking a functional form $f(\boldsymbol{x}; \vec{\theta})$ and determining the parameters $\vec{\theta}$ (also known as coefficients) at a set of input values $\boldsymbol{x}$ (possibly multi-dimensional) given a target set of expected outputs y. In other words, a regression algorithm is trying to find $\vec{\theta}^*$ where

$$\vec{\theta}^* = \arg\min_{\vec{\theta}} \text{RMSE}(f(\boldsymbol{x}; \vec{\theta}), y). \tag{2.5}$$

where RMSE stands for root mean squared error and can be computed as $\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$. Alternatively, other measures of error besides RMSE can be used to compute $\vec{\theta}^*$.

Any optimization algorithm can be used to perform regression. One such algorithm is the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm. L-BFGS-B is a variant of BFGS that allows for the use of bounds on $\vec{\theta}^*$.

2.3 Symbolic regression

In symbolic regression, like in regression, the goal is to find an equation that describes a given set of points y. However, symbolic regression does not require the functional form of the data to be known ahead of time. In other words, a symbolic regression algorithm is trying to find both $\vec{\theta}^*$ and f^* where

$$\left\{\vec{\theta}^*, f^*\right\} = \arg\min_{\vec{\theta},\, f} \text{RMSE}(f(\boldsymbol{x}; \vec{\theta}), y). \tag{2.6}$$

2.3.1 Genetic programming

One of the uses for genetic programming is symbolic regression. In the context of symbolic regression, genetic programming operates by first generating a population of equations (usually represented as trees) and then iteratively improving this population through mutation of the best equations as determined by some error metric (e.g. root mean squared error). In other words, genetic programming mimics natural selection in order to perform tasks including symbolic regression. [17]

2.3.2 Other existing methods

Although genetic programming is a popular algorithm for performing symbolic regression, there are other algorithms that are capable of performing symbolic regression including fast function extractor (FFX) [23] and Bayesian machine scientist [11].

Fast function extractor [23] is a non-evolutionary algorithm that performs symbolic regression. This algorithm combines many functions and determines which of those functions should be included in the final answer. The predicted function is a generalized linear model, which has the form $\hat{y} = a_0 + \sum_{i=1}^{N} a_i B_i(x)$ where the a_i's are coefficients and the B_i's are basis functions. Fast function extractor performs three steps (1) generate a large set of univariate and bivariate basis functions (e.g. $\{x, |x|, \log(x), x\log(x), \cdots\}$), (2) compute coefficients (a_i's) with decreasing regularization pressure in order to get the best coefficient sets for different numbers of non-zero coefficients, and (3) reduce the final set of solutions to a non-dominated pareto front that trades off between number

of basis functions (function complexity) and prediction error. If the final set of functions is larger than one, it is up to the user to determine which function is best.

The Bayesian machine scientist [11] is another non-evolutionary method that performs symbolic regression. In this approach, the researches created a corpus of closed-form equations (found on Wikipedia) and used a Markov chain Monte Carlo (MCMC) algorithm to move around the space of all closed-form equations. Using the corpus of equations and by formulating a probabilistic framework for the trade off between equation complexity and accuracy, the researchers are able to develop a prior and posterior distribution that can be used with the MCMC algorithm to both traverse the space of equations and to identify which equation that best explains a given symbolic regression problem.

2.4 y2eq

The authors of [3] created a convolutional sequence-to-sequence model [10] that is able to perform symbolic regression. The authors of [3] did not give their model a name, so in this thesis their model will be referred to as y2eq.

2.4.1 Data representation

The y2eq model is expected to solve many symbolic regression problems without update to the weights of the neural network between problems. To evaluate y2eq, y2eq receives a sequence of y-values that are ordered by x-values, which describe some unknown function f. Then, y2eq is expected

to output a sequence of one-hot vectors that represent tokens that are part of the functional form of the unknown function f. The functional form and the known y-values can now be used by an optimization algorithm ([3] uses BFGS) to determine which coefficients should be used in the functional form to approximate the y-values.

The dataset used to train such a function is formed such that each observation is a sequence of y-values paired with the underlying functional form as a sequence of one-hot vectors. In other words, a single observation in such a dataset is a single symbolic regression problem.

The ordering of the y-values is important since x-values are not provided to the neural network. Even if the x-values were provided to the neural network, it would still be desirable to keep the points ordered by x-values so that the convolutional layers of the encoder can extract information about the local behavior of adjacent points.

Chapter 3

Three challenges in symbolic regression

We have identified three important challenges in symbolic regression: corpus, coefficient, and cost. The challenge of devising a training corpus stems from the hierarchical nature of the data since the corpus should not be considered as a collection of equations but rather as a collection of functional forms and instances of those functional forms. The challenge of choosing appropriate coefficients for functional forms compounds the corpus challenge and presents further challenges during evaluation of trained models due to the potential for numerical similarity between instances of different functional forms. The challenge with cost functions (used to train the model) is mainly the choice between numeric cost and symbolic cost. All three challenges exist together and as a results are sometimes difficult to distinguish between. We will now define and describe these three challenges as best we can.

3.1 The corpus challenge

To train a neural network (or other method) to perform symbolic regression requires data. Symbolic regression is a task with abundant data because artificial datasets can always be generated: devise a collection of functions using some function-generating process, plug in x (univariate) or $\mathbf{x}$ (multivariate) values into each function, perhaps add some random noise to simulate measurement error, and now you have a set of training points for each known function from which we can train.

Being so straightforward, few authors have fully interrogated this process for deriving a training corpus. However, there are in fact subtle but distinct issues with devising a corpus in this manner.

The first issue is that there are many ways in which an equation can be written. For example, $x^2 + x$ can be written as $x(x + 1)$ or $x + x^2$ or $3x + x^2 - 2x$ and so on. If using symbolic cost (see Section 3.3) to evaluate predicted equations, one "correct" syntax for the equation must be stored as the target equation in the dataset. This means that a consistent way of writing equations must be established as was done in [3, 18]. For example, if a dataset contains the equations $x^2 + x$ and $(x + 1)(x + 2)$ (where the "correct" syntaxes are as written), then the trained model could potentially understand that the two equations are $x(x + 1)$ and $x^2 + 3x + 2$, but will be unfairly penalized for its choice of syntax. One benefit to this solution is that using a consistent syntax to write equations in the dataset will teach the model to write with the same consistent syntax; however, this solution is not entirely satisfactory as different syntaxes are useful in different situations. By using the consistent syntax, the model is likely unaware of other potentially useful syntaxes.

The second issue that the simple data generation technique fails to consider is the *hierarchical nature* of the training corpus. One does not have a collection of functions. Instead one has a collection of *functional forms* mixed together with a collection of *instances of functional forms*. Since we are dealing with a finite number of data points to express these instances of functional forms, there are actually infinitely many functional forms that can be fit to those points. In other words, there are many functional forms that can be considered a reasonable answer to any symbolic regression task. Given this knowledge, we must determine which functional forms and which instances of those functional forms to include in the corpus. Perhaps including only the simplest functional forms is the best approach; however, it may not always be easy to determine the simplest version of the functional form in question. So far, we have pointed out that instances of different functional forms that have similar y-values are problematic, but a single functional form that exhibits excessive differences in its behavior for different choices of parameters is also problematic.

Another issue involved in the corpus challenge is the choice of normalization. A neural network performs best on normalized data. So, if training a neural network to perform symbolic regression on many symbolic regression problems, then it is necessary to normalize the numeric data, namely the y-values. In [3], the y-values input to y2eq are normalized for each instance of each functional form separately. This maintains the shape of the functional form instances, but removes information about the scale of the y-values. Conversely, all y-values could be normalized together which would maintain the relative scale between different equations and maintain the shape of each equation, but many sets of normalized y-values would be virtually constant. For many tasks solved by neural networks, normalization is not difficult. Many

types of data used by neural networks (images, text, etc.) are contained in finite intervals of reasonable values; however, the outputs of arbitrary real valued functions is the set of all real numbers.

Another decision that needs to be made is which primitives to include in the corpus (and the model). In GP, for example, it is known that division is a problematic primitive [26, 14]. So far, in y2eq, division has not been used. Although, log is used for which all non-positive inputs are undefined.

3.2 The coefficient challenge

As we have already briefly described in the previous section, the corpus challenge is compounded by numerical coefficients. Let f_i be the i-th functional form in our training corpus. More properly, f_i will be a function of $\mathbf{x}$ and be parameterized by a vector of coefficients $\vec{\theta}$, i.e., $f_i(\mathbf{x}; \vec{\theta})$. For instance, if f_i is a simple linear function, we have $\mathbf{x} = x$, $\vec{\theta} = \left\{\theta^{(0)}, \theta^{(1)}\right\}$, and $f_i(x; \vec{\theta}) = \theta^{(0)} + \theta^{(1)}x$.

There are now an infinite number of functions for that functional form, following from changes to $\vec{\theta}$. It may also be important to prevent zeros in the elements of $\vec{\theta}$ as otherwise a constant function is part of the same functional form as the linear function (choose $\theta^{(1)} = 0$). This problem persists for small but non-zero $\theta^{(1)}$ (such as $\theta^{(1)} = 0.0001$). While this challenge is both a coefficient and corpus challenge, it is included here because it is important to consider zero (or small) coefficients in the evaluation phase as well as the data generation phase.

In [3], y2eq is paired with the optimization algorithm BFGS in order to fit coefficients to functional forms output by y2eq. In this partnership, it is

possible for y2eq to output a functional form that contains all possible terms and allow BFGS to zero out terms that are not necessary to explain the data.

This problem is not limited to the case of zeroing out terms. There are completely different coefficients that allow different functional forms to behave similarly. In other words, BFGS may find $\hat{\vec{\theta}}$ such that $\hat{f}_i(\boldsymbol{x}; \hat{\vec{\theta}}) \approx f_i(\boldsymbol{x}; \vec{\theta})$ where $\hat{f}_i \neq f_i$. For example, let $f_i(x; \vec{\theta}) = \theta^{(0)} e^x$ and $\hat{f}_i(x; \hat{\vec{\theta}}) = \hat{\theta}^{(0)} \sin(\hat{\theta}^{(1)} x + \hat{\theta}^{(2)}) + \hat{\theta}^{(3)}$. When $\vec{\theta} = (0.1)$ and $\hat{\vec{\theta}} = (-2.8533396, 0.46994925, 1.31815196, 3.)$, the root mean squared error is 0.057 for $x = \{0.1, 0.2, \cdots, 3.0\}$ (see plot in Figure 3.1).

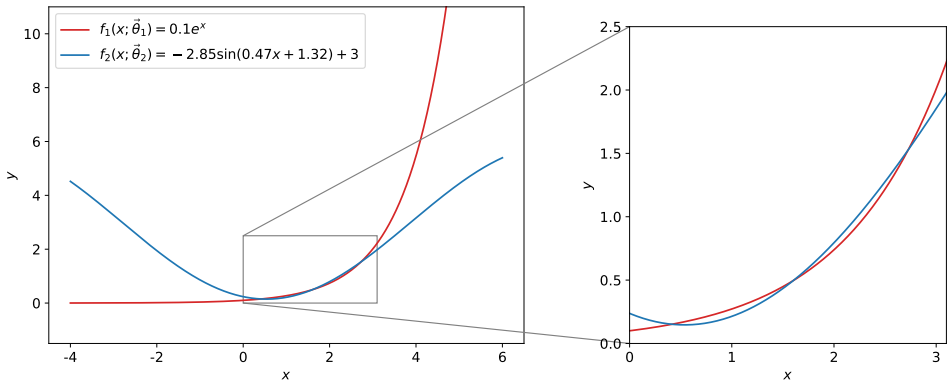

Figure 3.1: An example of instances of different functional forms that have similar y-values where the right plot is a zoomed in version of the plot on the left. The red curve is $f_i(x; \vec{\theta}) = \theta^{(0)} e^x$ and the blue curve is $\hat{f}_i(x; \hat{\vec{\theta}}) = \hat{\theta}^{(0)} \sin(\hat{\theta}^{(1)} x + \hat{\theta}^{(2)}) + \hat{\theta}^{(3)}$. When $\vec{\theta} = (0.1)$ and $\hat{\vec{\theta}} = (-2.8533396, 0.46994925, 1.31815196, 3.)$, the root mean squared error is 0.0571265890286456.

3.3 The cost challenge

A traditional (nonlinear) regression will be evaluated based on a fitting error, often the root mean squared error (RMSE): for training data $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$, let $\hat{f}$ be the estimated or fitted regression function. The root mean squared error is a numerical objective or *cost* function.

Numeric cost is also typically used in a symbolic regression context; however, symbolic regression is tasked with determining the functional form of $\hat{f}$ in addition to the coefficients. For this reason there is effort taken to ensure that $\hat{f}$ is a valid equation. If $\hat{f}$ were not a valid equation, then numeric cost would be undefined. For example, the sequence of mathematical tokens `x+` is not a valid equation because it is not clear how to compute the y-values.

There are a few drawbacks to the use of numeric cost in addition to its inability to evaluate invalid equations. Firstly, a low numeric cost can indicate either a well-fit solution of an overfit one. Second, in general, a numeric cost function is not amenable to backpropagation because the process of converting a representation of an equation to y-values is not differentiable – a necessary condition for backpropagation. For example, to compute numeric cost on the model y2eq, the model outputs a sequence of vectors that need to first be converted to mathematical tokens and then the tokens need to be combined to form an equation that then can be evaluated by root mean squared error. This conversion from a sequence of vectors to an equation is the part of the pipeline that prevents backpropagation. However, there are some methods that are able to backpropagate numeric cost in the symbolic regression domain such as [15], which uses primitives as activation functions in a neural network and then considers the neural network itself to be the equation.

Symbolic regression is concerned not just with matching the numeric values y, but also with being an accurate representation of the equation f assumed to be generating the data. Therefore, a cost function that compares f and $\hat{f}$ during training is beneficial. Such a *symbolic* cost function (that compares individual tokens) is easily amenable to backprogation via categorical cross

entropy, which can be written as

$$L_s(t, \hat{t}) = -\sum_{i=1}^{m} t_i \log\left(\text{softmax}\left(\hat{t}\right)_i\right) = -\sum_{i=1}^{m} t_i \log\left(\frac{e^{\hat{t}_i}}{\sum_{j=1}^{m} e^{\hat{t}_j}}\right) \tag{3.1}$$

where m is the number of tokens, t is the true token represented as a one-hot vector of length m, and $\hat{t}$ is the predicted token represented as a vector of length m.

The symbolic cost function is defined on invalid equations, but it can only be used to train a model that is trained on many symbolic regression tasks at once such as y2eq ([3]). The symbolic cost function requires the correct syntax of the underlying equation – of which there are many forms (see Section 3.1). But, because the correct syntax is specified, the model can learn to write equations in a more human way than some other algorithms like genetic programming [17] where code bloat is a common problem.

Both the numeric and symbolic aspects of symbolic regression are important concepts. Further, both numeric cost and symbolic cost have pros and cons. Combining numeric cost and symbolic cost during training should result in a successful model capable of symbolic regression. Later in this work, an idea for allowing y2eq to be trained with numeric cost and symbolic cost via backpropagation is presented (Section 4.3.3).

Chapter 4

Methods and Experimental Setup

The code used to generate equations, generate datasets, train models, evaluate models, run experiments, and analyze experiment results can be found on GitHub.[1]

4.1 Methods

In this section, details of the dataset used to train and evaluate the model and description of the model itself are written.

4.2 Dataset

The training dataset for y2eq consists of approximately 50 000 observations and the test dataset contains 1 000 observations. The dataset is hierarchical in nature because it uses both functional forms and specific instances of functional forms. There are 1 000 functional forms in the entire dataset and there are approximately 50 instances of each functional form in the training dataset (and

[1]Code available at https://github.com/rgrindle/y2eq.

only 1 instance of each functional form in the testing dataset). The instances of each functional form are used to compute the y-values that will act as input to the model while the functional form is the expected output. Notice that the model is not responsible for determining the coefficients of the functional form. Although, in some cases coefficients are determined by the optimization algorithm L-BFGS-B.

To generate such a dataset:

1. Generate a set of unique functional forms (using only the primitives in $\{+, \times, \wedge, \sin, \log, \exp\}$) which will be denoted as $\left\{f_i(x; \vec{\theta})\right\}_{i=1}^{1000}$, which is done using code from [3][2].

2. Select coefficients ($\vec{\theta}_{ij}$) randomly from a uniform distribution on the interval $[-3, 3]$ to create 50 instances of each functional form which will be denoted as $\left\{\left\{f_i(x; \vec{\theta}_{ij})\right\}_{i=1}^{1000}\right\}_{j=1}^{50}$.

3. Calculate the y-values at $x = \{0.1, 0.2, \cdots, 3.0\}$ for each instance of each functional form where $y_{ij} = f_i(x; \vec{\theta}_{ij})$ are the y-values of the j-th instances of the i-th functional form which are ordered by the x-values.

4. Normalize the y-values individually for each instance of each functional form by

$$n(y_{ij}, y) = \frac{y - \min y_{ij}}{\max y_{ij} - \min y_{ij}} \tag{4.1}$$

   where y are the y-values to be normalized and y_{ij} are the true y-values. For the purposes of creating the dataset $y = y_{ij}$, but during evaluation this will not be the case. In other words, we compute $\bar{y}_{ij} = n(y_{ij}, y_{ij})$.

[2]Original code: https://github.com/SymposiumOrganization/EQLearner, My version: https://github.com/rgrindle/y2eq

5. Store the dataset as $\{(\bar{y}_{ij}, T(f_i)) : \forall i = 1, 2, \cdots, 1000 \text{ and } j = 1, 2, \cdots, 50\}$ where $T(f_i)$ is the sequence of tokens used to write the functional form f_i. Like in [3], tokens are recorded as integers (see Table 4.1). There are more tokens here than are included in the target outputs. Note that the sequences of tokens are expected to be the same length so they are padded with the `PAD` token.

Table 4.1: Tokens and their integer ID

Token	Integer ID
`PAD`	0
x	1
sin	2
exp	3
log	4
(	5
)	6
$\wedge$	7
$\times$	8
$+$	9
$\div$	10
e	11
`START`	12
`END`	13
$\sqrt{\cdot}$	14
$-$	15
1	16
2	17
3	18
4	19
5	20
6	21
7	22
8	23
9	24

Generating functional forms

The code provided by the authors of [3] was used to generate 1 000 functional forms using the tokens: x, $\times$, $+$, sin, log, exp, (,), $\wedge$ and integers 1 through 9. Note log is the natural logarithm and the functional forms are written in infix notation. Integers are only used as exponents, not as coefficients.

Functional forms are generated first without considering placement of coefficients. Then, coefficients are placed in all possible locations except for exponents. For example, the algorithm could generate $x^6 + x^5 + 1$ and then the coefficients would be place like $\theta^{(1)}x^6 + \theta^{(2)}x^5 + \theta^{(3)}$.

Since the model will be trained using symbolic cost, the "correct" syntax for the functional form must be specified. There are many syntaxes for each functional form. For example, $x^2 + x = x(x + 1) = x + x^2 = x + x \times x = \cdots$. For the purpose of creating the dataset, each functional form is written in a consistent manner. This is achieved by using SymPy [24] to fully expand functional forms and order terms consistently. This is all done in [3].

However, in the original code there is one instance where inconsistent syntaxes are created. This has to do with polynomials in $\exp(x)$. For example, $f_1(x, \vec{\theta}_1) = \theta_1^{(1)} e^{\theta_1^{(2)} x}$ and $f_2(x, \vec{\theta}_2) = \theta_2^{(1)} e^{2\theta_2^{(2)} x}$ are semantically identical if $\theta_1^{(1)} = \theta_2^{(1)}$ and $\theta_1^{(2)} = 2\theta_2^{(2)}$.

This inconsistency occurs because of the exponent rule $e^{mx} = e^{x^m}$. More specifically, many functional forms are generated as polynomials in x, $\sin(x)$, $\log(x)$, or $\exp(x)$. Using the exponent rule, $\exp(x)^3 + \exp(x)^2 + \exp(x)$ can be written as $\exp(3x) + \exp(2x) + \exp(x)$ (which is how SymPy prefers to write this equation). This causes a problem because exponents can be modified by coefficients, which is not the case when using any of the other tokens.

To choose the "correct" functional form, we forced the exponent/coefficient to decrease moving left-to-right by one term at a time and to be the smallest possible integers. For example, $e^{4x} + e^{x}$ would become $e^{2x} + e^{x}$. After applying this rule to all functional forms it is easy to make a unique list of functional forms[3].

4.2.1 Generating instances of functional forms

Now that we have a list of functional forms, a list of instances of functional forms[4] is generated by applying coefficients that are chosen randomly from a uniform distribution on the interval $[-3, 3]$ to each functional form. This produces a list of instances of functional forms for which y-values are computed. An instance of a functional form is only included in the dataset if it has y-values such that for all y, $|y| \leq 1000$ and there are no invalid y-values (NaN's) as done in [3].

4.3 Models

Here the models that are used in the experiments conducted as a part of this thesis are discussed.

4.3.1 y2eq

Here we describe our re-implementation of the model from [3]. The authors of [3] do not give their model a specific name, but in this thesis it will be referred to as y2eq.

[3] Link to list of functional forms
[4] Link to list of instances of functional forms

y2eq is a convolutional sequence-to-sequence model [10], which receives 30 y-values as input and outputs a sequence of tokens (see possible tokens listed in Table 4.1). Figure 4.1 clearly indicates the inputs and outputs of y2eq. Notice that y2eq outputs the functional form as if all coefficients are ones and then the coefficients are placed where ever possible except as exponents.

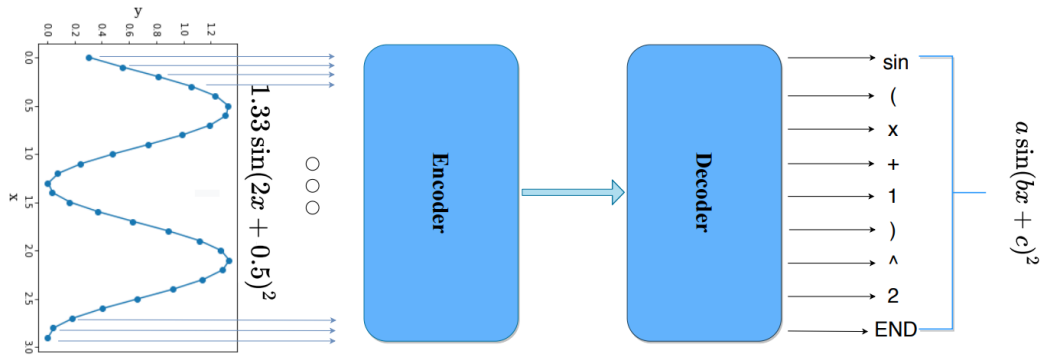

Figure 4.1: Depiction of y2eq. Image from [3].

Typically the input and output to a convolutional sequence-to-sequence model are both a sequence of tokens, which is represented as a sequence of vectors. On the input side, a common way to convert the tokens to their corresponding vectors is to use an embedding layer. This layer is a lookup table that takes the integer ID of a token (see Table 4.1) and produces a vector. But, this cannot be done with an embedding layer for y2eq because the input to y2eq is continuous rather than discrete. Thus, a fully-connected layer is used to create a vector for each y-value.

Table 4.2 identifies the hyper-parameters used in the architecture of y2eq.

Table 4.2: Hyper-parameters of y2eq with convolutional seq2seq architecture

Hyper-parameter	Value
Embedding dimension	256
Hidden dimension	512
Number of layers in encoder	10
Number of layers in decoder	10
Convolution kernel size in encoder	3
Convolution kernel size in decoder	3

4.3.2 y2eq-transformer

Transformers are the state-of-the-art architecture for sequence-to-sequence models. As a result, we have re-implemented y2eq as a transformer ([3] suggests the use of transformers in future versions of y2eq). The transformer, which will be referred to as y2eq-transformer, uses the hyper-parameters listed in Table 4.3.

Table 4.3: Transformer hyper-parameters used in y2eq-transformer and eq2y

Hyper-parameter	**Value**
Embedding size	512
Number of heads	8
Number of encoder layers	3
Number of decoder layers	3
Dimension of feed-forward layers	512
dropout	0.1

Like the non-transformer version of y2eq, the input to the neural network

is 30 floating point numbers, which is transformed to the embedding size of 512 (see Table 4.3) via a fully-connected layer (512 weights).

4.3.3 y2eq → eq2y (y2eq2y)

We have already discussed that using numerical cost to train y2eq via backpropagation is challenging because it is not clear how to differentiate the process of converting the output functional form to y-values. The purpose of y2eq2y is to use a neural network (eq2y) to make this conversion, then eq2y is a differentiable neural network and thus y2eq can be trained with numeric cost.

In Figure 4.2, y2eq2y is shown to be the combination of y2eq-transformer (Section 4.3.2) and eq2y (Section 4.3.3). It is important that eq2y is trained before y2eq so that its weights can be fixed during the training of y2eq. Otherwise, it may be possible for eq2y to output low error y-values even when y2eq does not output the correct functional form.

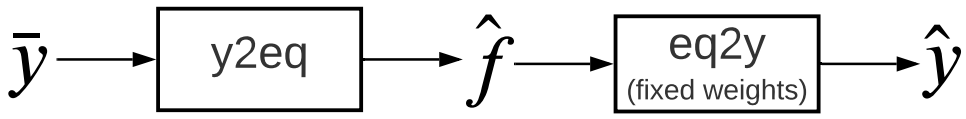

Figure 4.2: Architecture of y2eq2y

With this model, symbolic cost can still be used on the output of y2eq and now numeric cost can be used on the output of eq2y. These costs can be used separately or in combination; however, numeric cost should not be used on invalid equations because numeric cost is undefined for invalid equations. Thus, some amount of symbolic cost must be used during training. The combination

of the two costs can be written as

$$L(y, \hat{y}, T, \hat{T}, w) = \begin{cases} wL_n(y, \hat{y}) + (1-w)\frac{1}{|T|}\sum_{t\in T, \hat{t}\in\hat{T}} L_s(t, \hat{t}) & \text{if equation is valid} \\ \frac{1}{|T|}\sum_{t\in T, \hat{t}\in\hat{T}} L_s(t, \hat{t}) & \text{otherwise} \end{cases} \tag{4.2}$$

where y and $\hat{y}$ are the target and predicted y-values respectively, T and $\hat{T}$ are the target and predicted sequences of tokens, L_n (Equation 4.3) computes the numeric cost, L_s (Equation 3.1) computes the symbolic cost, and w is a weight in the interval $[0, 1)$.

eq2y

The model eq2y was designed to compute y-values for a given instance of a functional form. The dataset needed to train eq2y is similar to the dataset y2eq. In this dataset, input is the instance of a functional form and the output is the y-values. Notice that unlike with y2eq, the specific instances of the functional forms are used rather than the functional forms. Since eq2y will be used to take the output of y2eq as input, all instances of functional forms used for training will have coefficients of one.

Figure 4.3 shows the eq2y model architecture. Like y2eq, eq2y will always use the same x-values; however, eq2y receives the x-values. These x-values are input to the model where typically the previously output tokens are input. As a result the masked multi-head attention layer does not need to be masked. The x-values are used instead of the previously output y-values because doing so avoids the necessity for a `START` token in the sequence of y-values. Since y-values are continuous values rather than discrete, it is not clear how to incorporate a start token. Without a `START` token, there is nothing to input

when the model is expected to output the first y-value.

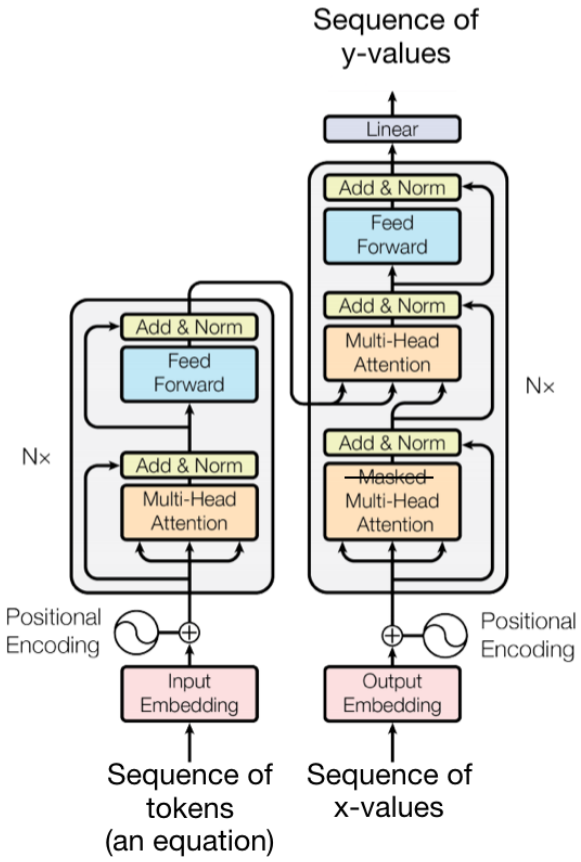

Figure 4.3: eq2y is a transformer that takes an equation (as a sequence of tokens) as input and produces a sequence of y-values. x-values are input to the model where normally the previously output tokens would be input. Image modified from [33].

4.4 Training models

There are two possible costs for training y2eq: symbolic cost and numeric cost. In symbolic cost (categorical cross entropy, Equation 3.1), each predicted token should match each target token exactly even though there are many syntaxes in which any equation can be written. In numeric cost (mean square error for example, Equation 4.3), the predicted equation and the y-values of

that equation would be computed then the numeric cost is the error between the predicted y-values and the target y-values.

As done by [3], we will be using symbolic cost. Symbolic cost allows us to use teacher forcing and backpropagation and maintain a well-defined cost function when the equations output by the neural network are not valid.

We do not generally expect symbolic cost to be used in regression algorithms. In fact, nearly all regression and symbolic regression algorithm discussed in Section 1 use numeric cost. However, as the authors of [3] have pointed out, numeric cost can lead to overfitting. They also say that symbolic cost can help teach the neural network to write equations consistently. For example, the equation $f(x) = 3x$ can be written as $f(x) = 5x - 2x$, but we prefer the first syntax. It is worth noting that symbolic cost still does not make sense for symbolic regression algorithms that only deal with one underlying equation at a time. Training parameters for all different models can be found in Table 4.4.

Table 4.4: Training hyper-parameters

Hyper-parameter	Value			
	y2eq	y2eq-transformer	eq2y	y2eq2y
Batch size	32	32	32	32
Optimizer	Adam	Adam	Adam	Adam
Learning rate	10^{-4}	3×10^{-4}	3×10^{-4}	3×10^{-4}
Gradient clipping	1	1	1	1
Number of epochs	105	287	-	-
Type of Loss	sym	sym	num	sym & num

4.5 Evaluating models

The models are evaluated using numeric cost on the testing dataset. Depending on the experiment, this may include the use of L-BFGS-B (see Section 4.5.1). The numeric cost is the RMSE of normalized y-values. This is done so that the errors are comparable between different equations. Otherwise equations with extremely large values would likely have large errors even if the model is doing a decent job of predicting the functional form. The normalization is done based on true y-values. So, to evaluate the model on the input $\bar{y}_{ij}$ with expected output f_i and known coefficients $\vec{\theta}_{ij}$, then y2eq outputs the predicted output $\hat{f}_i$ and we let $y_{ij} = f_i(x; \vec{\theta}_{ij})$ and $\hat{y}_{ij} = \hat{f}_i(x; \vec{\hat{\theta}})$. Using Equation 4.1, we compute the root mean squared error as $L_n(y_{ij}, \hat{y}_{ij})$ where

$$L_n(y, \hat{y}) = \text{RMSE}\left(n(y, y), n(y, \hat{y})\right). \tag{4.3}$$

Here the true y-values and the predicted y-values are both normalized by the parameters associated with the true y-values.

4.5.1 L-BFGS-B algorithm

L-BFGS-B is used in some of the experiments in this thesis. In those cases, L-BFGS-B attempts to find

$$\vec{\theta}^* = \arg\min_{\vec{\theta}} \text{RMSE}\left(y_{ij}, \hat{f}_i(x; \vec{\theta})\right). \tag{4.4}$$

Notice that the unnormalized y-values are used. Then, to evaluate the result, $\hat{y} = \hat{f}_i(x; \vec{\theta}^*)$ and $x = \{0.1, 0.2, \cdots, 3.0\}$ are used in Equation 4.3.

L-BFGS-B requires initial guesses for the coefficients, thus the algorithm may terminate with different results for different choices of initial guesses. To try to make the results more consistent, we run L-BFGS-B 150 times with randomly chosen initial coefficients from the uniform distribution on the interval $[-3, 3]$. We also provide bounds to L-BFGS-B to ensure that all coefficients are in $[-3, 3]$ to help avoid overfitting.

In Figure 4.4, some evidence is provided to show that 150 random restarts of L-BFGS-B is a reasonable choice. Here the numeric cost (Equation 4.3) of the best coefficients predicted by L-BFGS-B has been computed for each observation in the testing dataset for various numbers of random restarts. Since the RMSE shows little to no improvement when the number of random restarts is above 150, 150 random restarts was chosen to be used when evaluating the models.

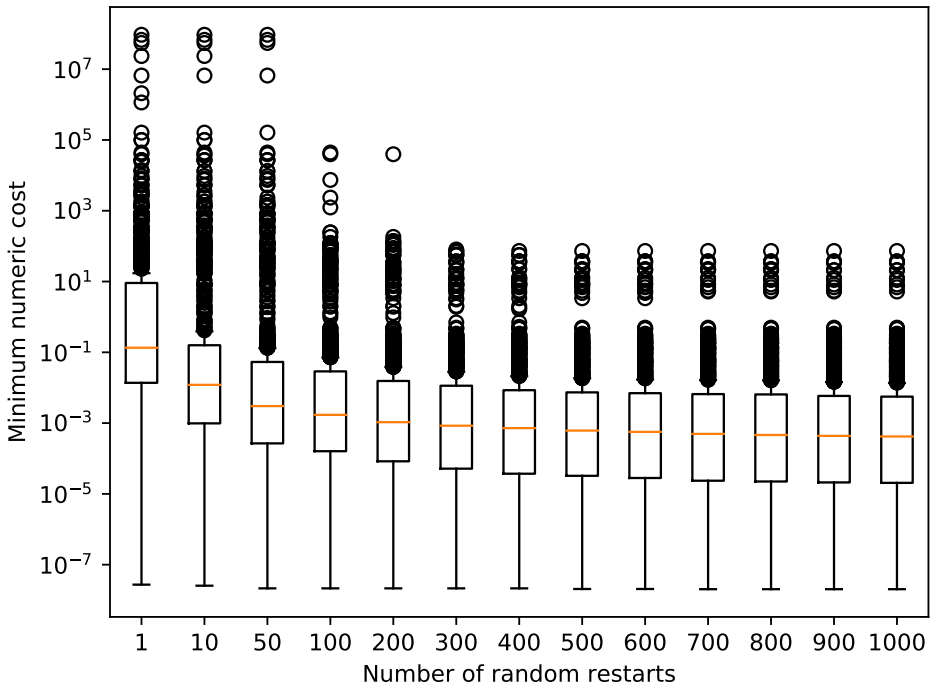

Figure 4.4: Performance of L-BFGS-B for various numbers of random restarts. Each boxplot contains 1 000 data points (one for each observation in the test dataset). The RMSE decreases as the number of random restarts increase, but there is not much difference in RMSE between 150 and 1 000 random restarts. As a result, we have chosen to use 150 random restarts when evaluating the model.

4.6 Experimental Setup

First, we show that we have successfully recreated the model from [3] and that model is able to achieve lower root mean squared errors than GP in the extrapolation region ($x_{\text{ext}} = [3.1, 3.2, \cdots, 6.0]$).

Then, we show evidence of the three challenges in symbolic regression: corpus, coefficient, and cost.

4.7 y2eq

In this paper, the y2eq model will be trained for 105 epochs with a batch size of 32 using the optimizer Adam [16] with a learning rate of 10^{-4}. The model has also been trained with dropout [31] and gradient clipping.

For all experiments, y2eq (or any of its variants) use the model parameters specified in Section 4.3 and the training parameters specified in Table 4.4. There are a few possible differences in these experiments: training dataset, testing dataset, use of L-BFGS-B or not[5]. Any of these experiment's specific alterations are mentioned in Section 5 right next to the results.

4.8 Numeric regression

In [3], a neural network that maps x to y for a single underlying function was used as control against y2eq. This neural network consists of 3 fully-connected layers of 100 units each using the ReLU activation function on all hidden layers and an identity activation function on the units in the output layer.

The numeric regression neural network does not result in a compactly written symbolic equation as expected of a symbolic regression algorithm, so it is important to use a true symbolic regression algorithm as control in addition to the numeric regression neural network. We will use genetic programming as that control.

[5]When we use L-BFGS-B we use the SciPy [34] implementation of L-BFGS-B

4.9 Genetic programming

Genetic programming (GP) is used only for the first experiment that compares y2eq, GP and numeric neural networks. Genetic programming was performed with the following parameters. Following convention, initial random trees were restricted to a depth of six, ramped-half-and-half was used to initialize the population, evolved trees were restricted to a depth of 17, and we employed a population size of 100. Each GP run terminates after 100 generations.

The primitive set is $\{\times, +, \sin, \log, \exp, (\cdot)^2, (\cdot)^3, (\cdot)^4, (\cdot)^5, (\cdot)^6\}$ and the terminal set is $\{x\}$. These primitives and terminals correspond to the tokens available to the neural network. Note that protected versions of primitives were used for primitives that can produce undefined or infinite output. These are all primitives expect $\times$ and $+$.

Before a GP tree is evaluated, coefficients are determined using the nonlinear optimization algorithm L-BFGS-B [36] (we used the SciPy [34] implementation). Then, the RMSE is computed between the equations with coefficients and the target y-values. Since coefficients are calculated and can be negative, subtraction is not needed in the primitive set.

4.10 Comparing y2eq, GP and numeric NN

The y2eq model, genetic programming, and the numeric regression neural network will be compared by root mean squared error in the extrapolation

region. The extrapolation region is located at $x_{\text{ext}} = \{3.1, 3.2, \cdots, 6.0\}$. The way that these algorithms are compared is described below.

The y2eq model is evaluated as follows. First, normalized, ordered y-values are input to the neural network and the neural network outputs a sequence of tokens that should represent a functional form. Second, unnormalized y-values from the interpolation region ($x_{\text{int}} = \{0.1, 0.2, \cdots, 3.0\}$) and the functional form are used by L-BFGS-B to pick coefficients for the functional form to minimize root mean squared error in the interpolation region: $[0.1, 3.1)$. To compare the root mean squared error between different true underlying equations, Equation 4.3 is used on the extrapolation region ($x_{\text{ext}} = \{3.1, 3.2, \cdots, 6.0\}$).

In the numeric regression neural network, the network is trained to output the normalized y-values in the interpolation region. Since this neural network does not output an equation, the predicted y-values in the extrapolation region are calculated by inputting the x-values in the extrapolation region to the neural network. The output of the neural network is y-values for the extrapolation region that have been normalized based on the y-values in the interpolation region, so the RMSE is calculated between these normalized outputs and the expected outputs, which have also been normalized.

In genetic programming, the unnormalized y-values are provided as targets so that nonlinear regression can be performed without normalization (just like for y2eq). Then, the true and predicted y-values are normalized based on the true y-value to compute the root mean squared error (Equation 4.3). The equation/tree with the lowest root mean squared error on the validation points is considered the best equation. This best equation outputs are normalized and used when calculating the root mean squared error in the extrapolation region.

Genetic programming and the numeric regression neural network are not capable of transfer learning like y2eq, so GP and the numeric NN are trained for each symbolic regression problem that they are compared with y2eq while y2eq is trained for all problems at once, but is never exposed to the exact functional form instances until test time.

Chapter 5

Results

Before discussing results relating to the corpus, coefficient, and cost challenges, we will show that y2eq has been successfully recreated from [3]. Figure 5.1 shows that y2eq (see Section 4.3.1) outperforms numeric neural networks (see Section 4.8), and genetic programming (see Section 4.9) in the extrapolation region, $x_{\text{ext}} = \{3.1, 3.2, \cdots, 6.0\}$, in terms of numeric cost (see Equation 4.3). In this experiment L-BFGS-B is used with a single initial guess, which is chosen randomly from the interval $[-3, 3]$. Using 150 random restarts (as discussed in Section 4.5.1), would mean considerably longer run times for GP.

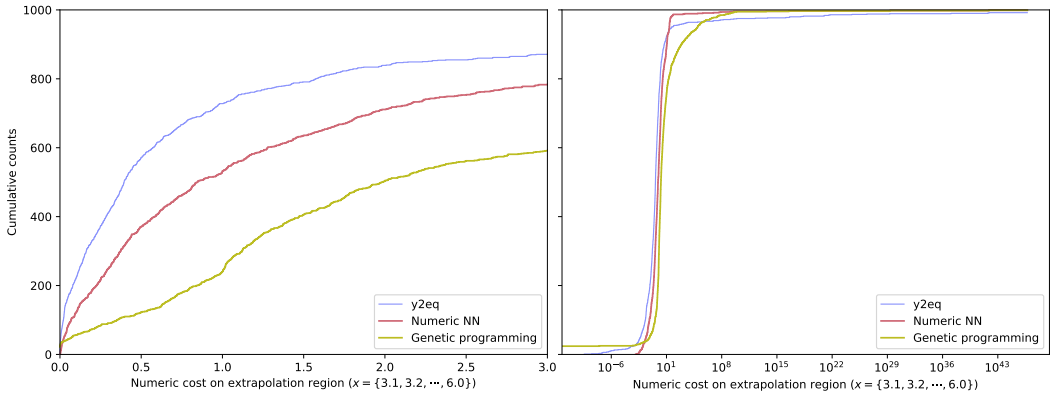

Figure 5.1: This figure shows the recreation of the result from [3]. The authors of [3] only show the left plot, which has linear scaling and only shows numeric cost in $[0, 3]$. The right plot displays the same data in log scale and no restriction on numeric cost. Here all numeric costs are computed as described in Equation 4.3 with the use of L-BFGS-B. Another addition from the results of [3] is the inclusion of genetic programming (which also uses L-BFGS-B). By using a Mann-Whitney U test we can present statistical statistical significance, after a Bonferroni correction, that indicate y2eq achieves lower numeric cost than a numerical neural network ($U = 369029.0$, $p < 10^{-22}$) and y2eq achieves lower numeric cost than genetic programming ($U = 222893.0$, $p < 10^{-100}$) in the extrapolation region.

5.1 The corpus challenge

The challenge of devising a training corpus stems from the hierarchical nature of the data since the corpus should not be considered as a collection of equations but rather as a collection of functional forms and instances of those functional forms.

Many unique functional forms can produce similar y-values with certain coefficients. When these instances of functional forms are in the dataset, it causes conflicting costs. To demonstrate this, we created a dataset where instances of functional forms have been purposely picked to have similar y-values to other observations in the dataset. This dataset will be referred to as the confusing dataset. Figure 5.2 shows a comparison of y2eq-transformer

when trained using the typical dataset (see Section 4.2) and the confusing dataset. This result shows that y2eq-transformer produces slightly higher RMSE when trained on the confusing dataset; however the normal dataset has some large outliers.

Normally, the dataset is created by randomly choosing functional forms and coefficients for those functional forms, but to create the confusing dataset the coefficients are calculated. The confusing dataset can be constructed using the following instructions.

1. Generate a fourth of the typical dataset (instruction in Section 4.2) by using a fourth of the functional forms. Call this fourth of the dataset $D_{1/4}$. Call the set of functional forms used $F_{1/4}$ and the set of unused functional forms $F_{3/4}$.

2. To generate the rest of the dataset,

   (a) Randomly choose a functional form $f \in F_{3/4}$.

   (b) Pick the unnormalized y-values from a random observation of the dataset $y \in D_{1/4}$

   (c) Choose the coefficients $\vec{\theta}$ so that $f(x;\vec{\theta}) \approx y$ by using L-BFGS-B.

   (d) Include the newly created observation $\left(f(x;\vec{\theta}), T(f)\right)$ in the full dataset. The full dataset includes $D_{1/4}$ and all new observations created by this step. Recall that $T(f)$ is the sequence of tokens used to write the functional form f.

   (e) Repeat step (2) until the dataset is complete.

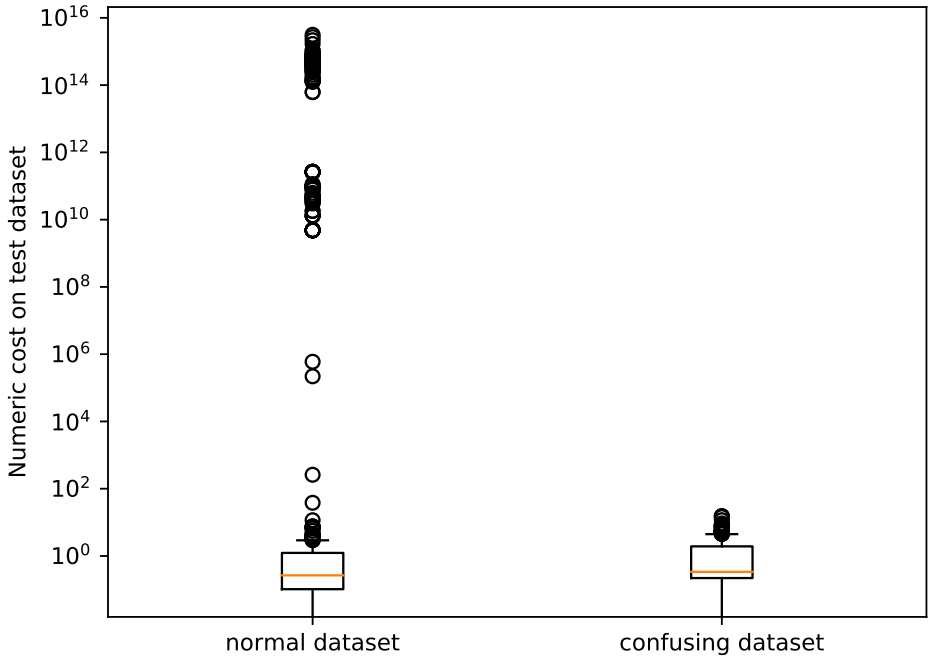

Figure 5.2: The model y2eq-transformer produces greater numeric cost when trained on a dataset that was built to contain many instances of functional forms that are numerically similar to different functional forms. Note that in both treatments the models were evaluated on a test dataset where all coefficients are ones and thus L-BFGS-B was not used (see Section 5.2). This result holds statistical significance from the Mann-Whitney U test ($U = 573931.5$, $p < 10^{-15}$).

Next, we show that accumulating instances of different functional forms with similar y-values in the dataset is easy to do by accident. The dataset that has been created by picking functional forms and coefficients randomly (see Section 4.2) has been searched for pairs of observations in the dataset that contain similar y-values. The 100 smallest root mean squared errors between different observations (specifically the y-values) have been recorded. Of those 100 comparisons, there are 75 comparisons between different functional forms. These 75 comparisons are plotted in Figure 5.3 and Figure 5.4. (These figures are separate only because a combined figure would not fit on a single page.)

In all these comparisons, the RMSE is about 10^{-5} and there is no real visible difference between the plotted y-values. Each pair of functional forms (in the legend of each subplot) contain shared terms.

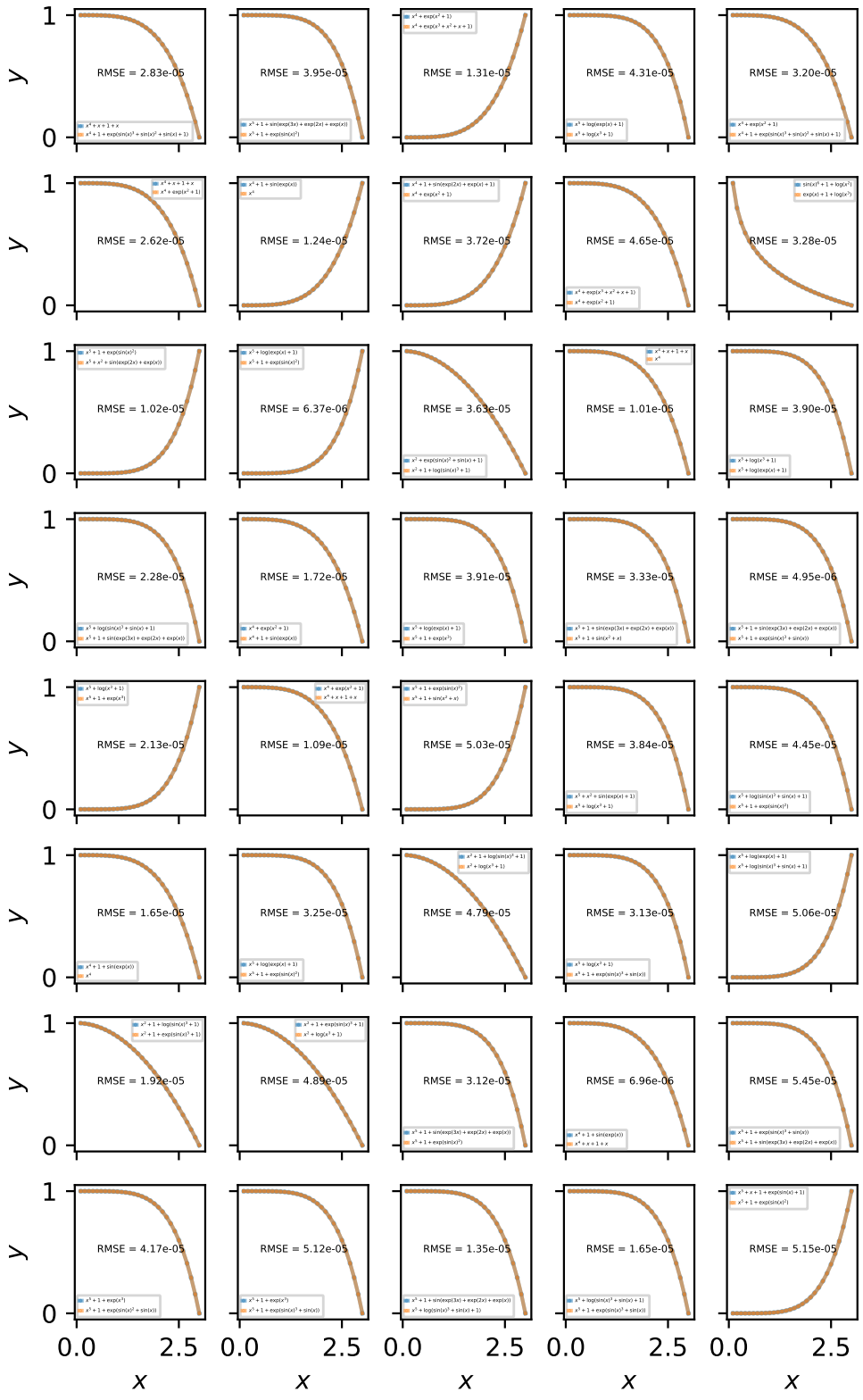

Figure 5.3: Plots of different functional forms that result in similar y-values for at least one functional form instance in the dataset.

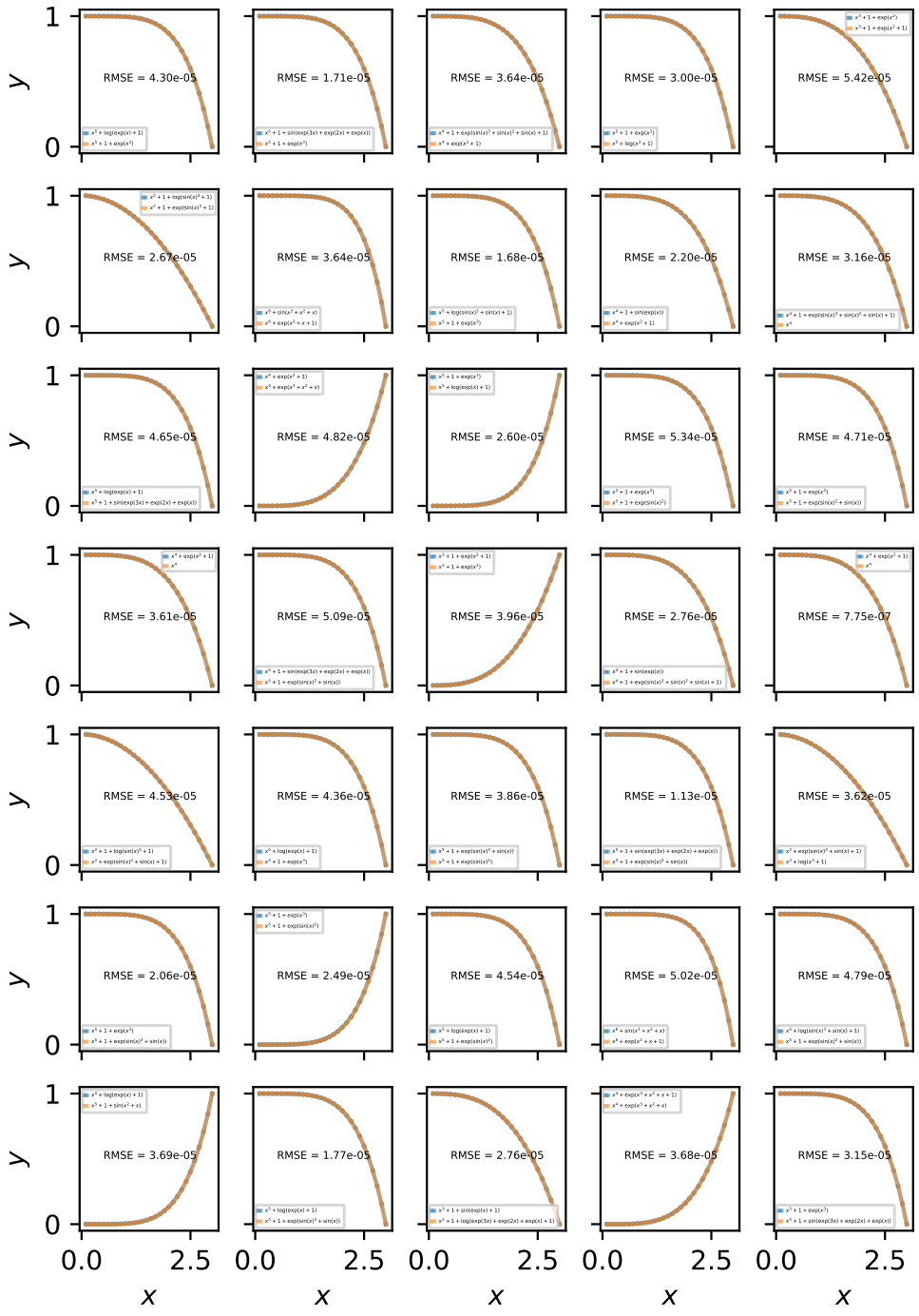

Figure 5.4: More plots of different functional forms that result in similar y-values for at least one functional form instance in the dataset.

For each of these plots we have two functional form instances ($f_i(x; \vec{\theta}_i)$

and $f_j(x;\vec{\theta}_j)$) that are numerically similar. We expect that the reason for their similarity is that the shared terms are dominant terms. To test this, we compute the RMSE between $f_i(x;\vec{\theta}_i)$ and its shared term only version. (We also compute RMSE for $f_j(x;\vec{\theta}_j)$ and its shared term only version.)

For example, take the top left plot of Figure 5.3 where the two functional forms are $f_i(x;\vec{\theta}_i) = \theta_i^{(1)}x^4 + \theta_i^{(2)}x + \theta_i^{(3)}1 + \theta_i^{(4)}x$ and $f_j(x;\vec{\theta}_j) = \theta_j^{(1)}x^4 + \theta_j^{(2)}1 + \theta_j^{(3)}\exp(\theta_j^{(4)}\sin(\theta_j^{(5)}x)^3 + \theta_j^{(6)}\sin(\theta_j^{(7)}x)^2 + \theta_j^{(8)}\sin(\theta_j^{(9)}x) + \theta_j^{(10)}1)$ which have shared terms x^4 and 1. Let $g(x;\vec{\theta}) = \theta^{(1)}x^4 + \theta^{(2)}1$. Thus, the shared term version of $f_i(x;\vec{\theta}_i)$ is $g(x;(\theta_i^{(1)},\theta_i^{(3)}))$ and the shared term version of $f_j(x;\vec{\theta}_j)$ is $g(x;(\theta_j^{(1)},\theta_j^{(2)}))$.

Figure 5.5 shows the RMSE between functional forms instances and their shared terms versions is small (although not as small as the RMSE between the two instances of the functional forms). This indicates that the shared terms dominate the functional form instances.

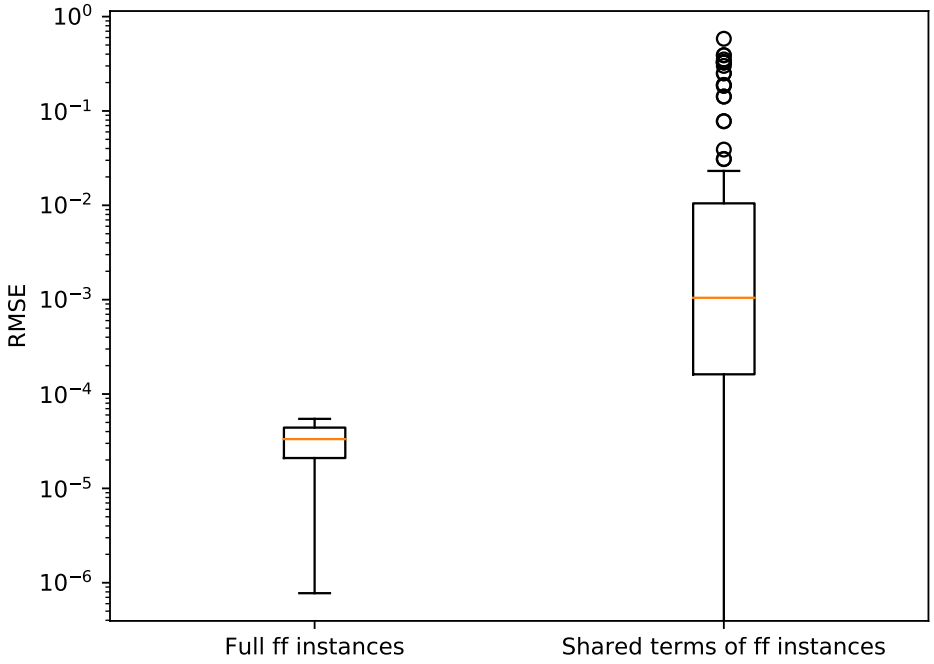

Figure 5.5: The left boxplot contains the root mean squared errors between the functional form instances of Figures 5.3 and Figure 5.4. The right boxplot shows the RMSE between the full functional forms and their shared term versions of these functional form instances. Both boxplots show small RMSE, so there are numerically similar functional form instances between different functional forms in the existing dataset and the shared terms (or just some of them) of these numerically similar functional forms instances are dominant terms that explain most of the behavior of the functional form instances.

5.2 The coefficient challenge

The challenge of choosing appropriate coefficients for functional forms compounds the corpus challenge and presents further challenges during evaluation of trained models due to the potential for similarity between instances of different functional forms.

In Section 3.1 and Section 3.2, there are examples of functional form in-

stances where coefficients exist or are chosen for functional forms that result in different functional forms with similar y-values. This has been discussed in the previous section as a challenge related to dataset generation, but now it will discussed as a challenge in the evaluation phase. In [3], BFGS is used to compute coefficients of functional forms output by y2eq-transformer. Here we show that low numeric cost when using L-BFGS-B with y2eq-transformer can lead us to falsely believe that y2eq-transformer is accurately predicting functional forms.

If y2eq-transformer is able to predict functional forms correctly, it should perform well with and without the use of L-BFGS-B on a test dataset where all coefficients used to generate the y-values are ones. For such a dataset, it is possible for y2eq-transformer to write the exact instance of the functional forms without using L-BFGS-B, so using L-BFGS-B should not reduce the error of the output equations. Figure 5.6 shows that y2eq-transformer achieves lower numeric cost when using L-BFGS-B indicating that y2eq-transformer is not good at determining functional forms.

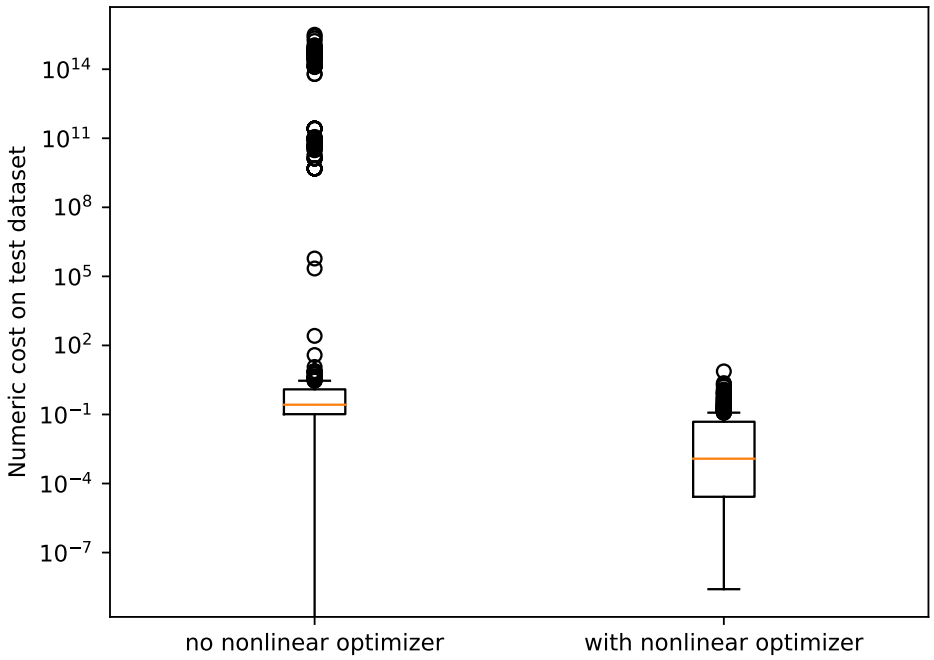

Figure 5.6: Here y2eq-transformer is evaluated with and without the use L-BFGS-B on a test dataset where all coefficients in functional forms instances are ones. y2eq-transformer receives higher numeric cost when L-BFGS-B is not used. There is statistical significance for this experiment due to a Mann-Whitney U test ($U = 862611.0$, $p < 10^{-100}$).

Now that we know y2eq-transformer does not predict functional forms accurately, we begin to look for explanations. One possibility is that y2eq-transformer outputs overly complex functional forms and used L-BFGS-B to zero the unnecessary terms. Figure 5.7 shows that y2eq-transformer outputs longer equations than the ones used to create the dataset.

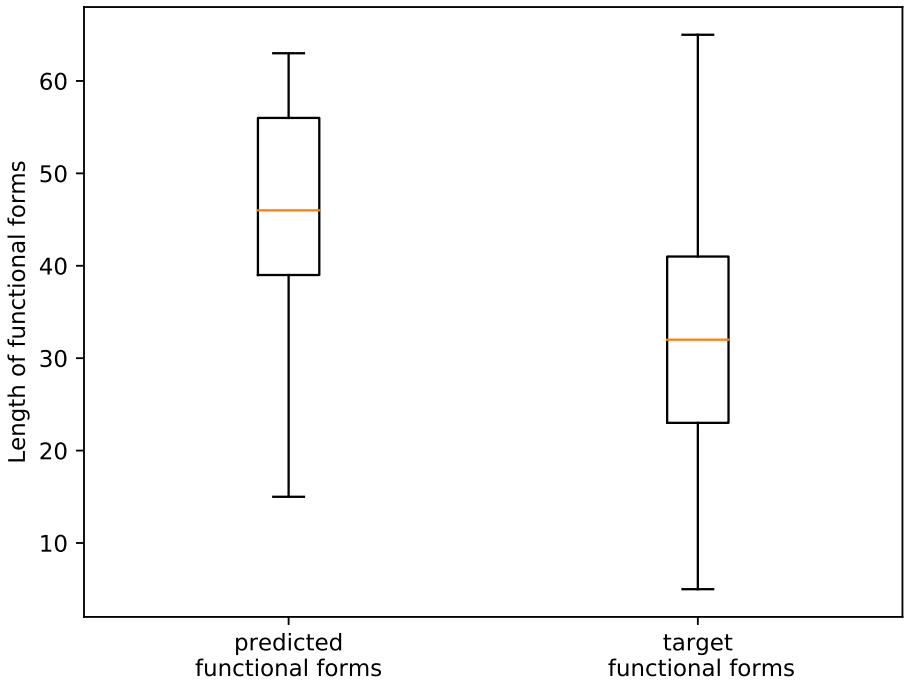

Figure 5.7: y2eq-transformer produces longer equations than those used to create the test dataset. There is statistical significance to support this claim from the Mann-Whitney U test ($U = 805927.0$, $p < 10^{-100}$).

5.3 The cost challenge

In Section 5.2, we have shown that y2eq-transformer produces larger functional forms than the functional forms used to generate the dataset. At first glance this result does not make sense because the symbolic cost should be penalizing functional forms that are longer than the target functional form. Furthermore, y2eq-transformer is not aware that it may be allowed to use L-BFGS-B during testing.

The following examples (Table 5.1 and Table 5.2) illustrate that the common technique of only counting cost when the target token is not `PAD` (a

padding token), creates a bias toward longer equations when the left-most part of the equation is correct. In Table 5.1, the target functional form is shorter than the predicted functional form and the symbolic cost only penalizes a single incorrect token (because when the target token is PAD, cost is ignored). In Table 5.1 (and Table 5.2), green indicates a target token that was not correctly predicted, red indicates a predicted token that is incorrect that is penalized, and blue indicates incorrectly predicted tokens that are not penalized.

Table 5.1: Overly long predicted functional forms are only partially penalized

target	log(x)^3+log(x)^2END	PAD	PAD	PAD	PAD	PAD	···
prediction	log(x)^3+log(x)^2+	log	(	x	)	END	···

Since symbolic cost does not compute cost when the target token is PAD, the prediction in Table 5.1 is only penalized for a single wrong token (+ instead of END) even though the prediction included 4 tokens (+ log (x)) that where not meant to be predicted.

In Table 5.2, the target and the prediction have been switched. Now, the target functional form is longer than the predicted functional form. This time, y2eq is penalized for all 6 incorrectly placed tokens (END PAD PAD PAD PAD PAD instead of + log (x) END) in its prediction by symbolic cost.

Table 5.2: Predicted functional forms that are too short are fully penalized

target	log(x)^3+log(x)^2+	log	(	x	)	END	PAD	···
prediction	log(x)^3+log(x)^2END	PAD	PAD	PAD	PAD	PAD	PAD	···

In Table 5.1 and Table 5.2 we have seen that y2eq-transformer is penalized more harshly for predicting an equation that is too short than it is for

predicting an equation that is too long. In other words, there is bias toward predicting longer functional forms. Notice that these examples assume that the beginning of the functional forms are identical.

In Figure 5.8, y2eq-transfomer has been trained in two different ways: (1) when symbolic cost is applied to every token (y2eq-transformer-pad) and (2) when symbolic cost is only applied when the target token is not `PAD` (y2eq-transformer). The results are that y2eq-transformer-pad produced smaller functional forms than y2eq-transformer, but not as small as the functional forms used to generate the dataset.

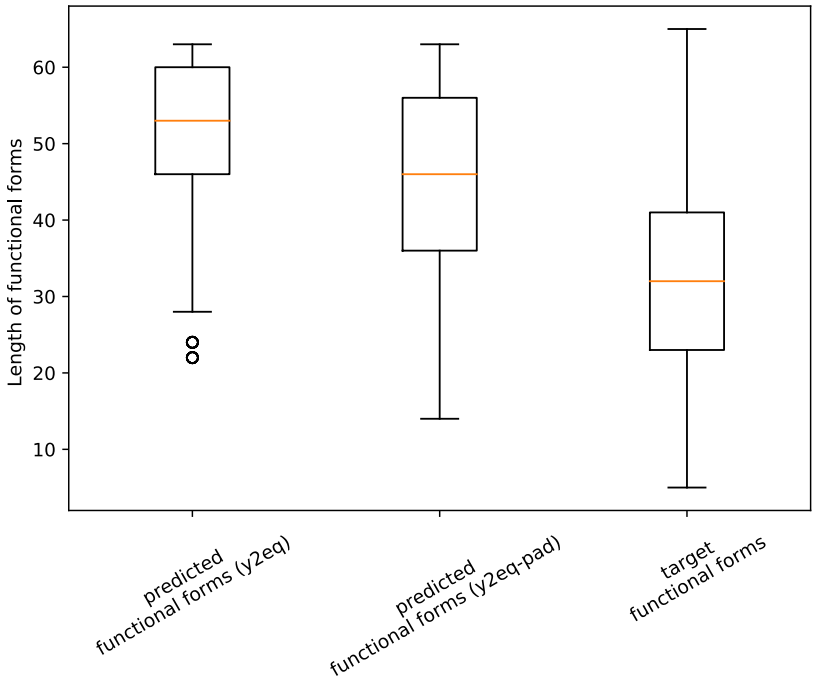

Figure 5.8: For the following results there is statistical significance provided by Mann-Whitney U tests. These statistical tests are still significant after a Bonferroni correction. y2eq-transformer produces longer equations than y2eq-transformer-pad ($U = 354277.0$, $p < 10^{-20}$) and y2eq-transformer-pad produced longer equations than the equations used to created the datasets on which both these models have been evaluated ($U = 710010.5$, $p < 10^{-80}$).

In Figure 5.9, y2eq-transformer and y2eq-transformer-pad are compared again. This time, y2eq-transformer-pad does not have lower numeric cost than y2eq-transformer.

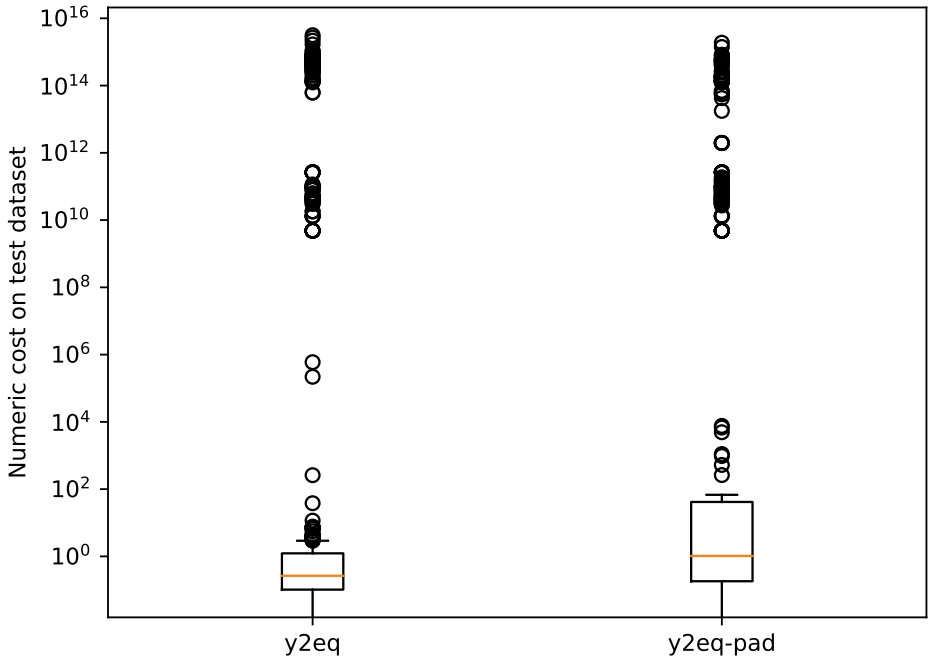

Figure 5.9: For the following result, there is statistical significance provided by Mann-Whitney U tests. y2eq-transformer produces lower numeric cost than y2eq-transformer-pad ($U = 407312.0, p < 10^{-5}$). These models are evaluated without any nonlinear optimizer on the test dataset that uses only ones for coefficients because of the coefficient challenge (see Section 5.2).

Chapter 6

Conclusions

This work has shown that the coefficient, corpus, and cost challenges are all present in symbolic regression. Specifically we have shown that

1. Training y2eq-transformer on a dataset that was explicitly designed to contain many instances of different functional forms that have similar y-values increases numeric cost compared with a model trained on the typical dataset (see Section 4.2).

2. In the typical dataset (see Section 4.2), there already are many (at least 75) instances of different functional forms that have similar y-values.

3. y2eq-transformer achieves lower numeric cost when allowed to use L-BFGS-B, indicating that y2eq-transformer is not able to accurately predict functional forms.

4. y2eq-transformer produces overly complex functional forms.

5. Since symbolic cost is only applied when the target token is not `PAD`, the symbolic cost contains a bias toward long equations provided that the beginnings of the equations match.

6. Altering symbolic cost to be applied to every token (even when the target token is `PAD`) results in a model (y2eq-transformer-pad) that produces shorter functional forms than y2eq-transformer but longer functional forms than necessary. Unfortunately, y2eq-transformer-pad does not achieve lower numeric cost than y2eq-transformer (without the use of L-BFGS-B).

6.1 Issues not addressed

Beyond the three challenges that have been the focus of this work, there are several other issues with y2eq that will eventually need to be addressed. (Many of these were originally mentioned in [3].) Issues not addressed in this work include:

1. y2eq relies on fixed x-values.
2. y2eq currently only handles univariate functional forms.
3. Unlike other symbolic regression algorithms, y2eq only produces a single functional form. Perhaps, y2eq would benefit from the ability to iteratively improve the predicted functional form as done by other symbolic regression algorithms.

6.1.1 Freeing x-values

Currently the models presented in this work only see y-values at fixed x-values, specifically $x = \{0.1, 0.2, \cdots, 3.0\}$. One way that these x-values can be freed is to continue to contain them in an interval $[0.1, 3]$, but change the

precise values. The x-values can be altered by pulling them from various types of distributions such as uniform or normal distributions. The x-values could be given even more freedom by allowing $x \in \mathbb{R}$ rather than just $x \in [0.1, 3]$.

The fact that y2eq does not receive x-values as input is a potential problem. If differences in x-values are allowed, but y2eq is unable to detect these differences then it seems unlikely that y2eq would be able to successfully predict the underlying functional form.

The inclusion of differences in x-values adds to the corpus and coefficient challenges. For example, functional forms with coefficients that perform horizontal shifts can have exactly the same y-values for different x-values or coefficients.

If y2eq is unable to adapt to different x-values, then y2eq will not generally be a useful algorithm for real world symbolic regression problems because regression datasets come with all sorts of different inputs.

6.1.2 Multivariate functional forms

The difficulties discussed in regard to different x-values is compounded when moving from univariate to multivariate functional forms. This is because the outputs (y-values) have a clear order for univariate functional forms, but not for multivariate functional forms. In the univariate case, there are only two directions along which one can travel (the $-x$ direction or the $+x$ direction), but in the multivariate case, there are infinitely many directions (along any vector in $\mathbb{R}^2$). Further, the input data ($\boldsymbol{x}$-values) in the multivariate case, are likely not in a single direction, meaning that not all nearby points in the numerical representation of the functional form can be next to each other in

the one dimensional sequence of y-values that are input to y2eq.

Perhaps a set-to-sequence type architecture (such as [29, 6]) can be utilized to overcome some of these difficulties. Alternatively, perhaps thinking of the input as an image (or a plot) in as high a number of dimensions as necessary and utilizing a model architecture used for the task of image captioning such as [35, 9] can give y2eq information about $\boldsymbol{x}$-values.

Like with the x-value problem, if y2eq cannot be scaled to multivariate symbolic regression problems, then y2eq will, in many cases, not be useful for solving real world symbolic regression problems because many real world regression problems are multivariate.

6.1.3 Iterative y2eq

The models presented in this work are the only models we are aware of that perform symbolic regression in a single attempt (only one equation is output by y2eq for a single set of y-values). All other methods iteratively improve equations to better fit the underlying data. We suspect that providing the ability for these models to iteratively improve their initial functional form would increase the performance of the models.

Such a model would receive numerical data (either y-values or errors) and symbolic data (the previously predicted equation). Perhaps a co-attention mechanisms (like [22]) can be useful to relate the symbolic and numeric data in these models.

Perhaps such models could even work in groups. For example, train a population of iterative y2eq models and have each generate an equation. Then, all (or a subset) of the equations can be input to the population of iterative y2eq

models to produce the next set of equations. This process can be thought of as using iterative y2eq models as the mutation operation in genetic programming.

6.2 Future work

Despite the importance of the issues not addressed (previous section), we believe that the three challenges described in this work should be solved first.

6.2.1 The corpus challenge

In this work, we have explored the similarity between y-values of instances of different functional forms and determined that their presence in the training dataset negatively effects the final performance of y2eq. We have also found that these instances of different functional forms with similar y-values are present in a dataset that was generated by randomly choosing coefficients and functional forms. At this point it is unclear how to remove these confusing instances of functional forms, but an experiment where only some of the confusing functional forms are removed could provide insight into the usefulness of removing all confusing instances of functional forms.

Perhaps the discovery that many of the instance of different functional forms that are numerically similar share dominant terms suggests that all the non-dominant terms should not be included in the target functional form. In other words, choose the simplest functional form to be the correct one. While we think this is a good idea, it is not always easy to know which functional forms are simplest.

So far, we have been focused on numerically similar functional form in-

stances that do not share the same functional forms, but perhaps it is also worth investigating functional forms that have wildly different behavior for different choices of coefficients. Perhaps these functional forms should be excluded from the dataset. More specifically, we can describe these functional forms as having large V for fixed x where

$$V(f_i, x, \vec{\theta}) = \max_{d\vec{\theta}} \left\| f_i(x, \vec{\theta}) - f_i(x, \vec{\theta} + d\vec{\theta}) \right\|. \tag{6.1}$$

It seems likely that a functional form that produces a large $V(f_i, x, \vec{\theta})$ will require more instances to be included in the dataset to allow the model to fully understand the functional form.

Perhaps there is also something to say about functional forms that have "hard-to-find" regions of $\vec{\theta}$ where the functional form has larger $V(f_i, x, \vec{\theta})$. There are many more questions that can be explored to try to identify problematic function forms and their instances.

Instead of proposing many possible ways to identify instances of functional forms that are problematic, perhaps the corpus challenge can be overcome by borrowing an idea from the field of active learning. That is, train on a portion of the dataset and identify problematic instances of functional forms. After identifying some of the problematic equations, perhaps the characteristics that make these functional forms problematic can be determined. If so, hopefully this knowledge can be incorporated into the generation of future datasets.

6.2.2 The coefficient challenge

In pursuit of solutions to the coefficient challenge, the best we have managed so far is to simply not use L-BFGS-B (or any other nonlinear optimiza-

tion algorithm) to determine the coefficients. This eliminates confusion about how well predicted functional forms compare to the target functional forms. However, in y2eq's current form, y2eq can only output one instance of each functional form (the instance where all coefficients are ones) when y2eq is not allowed to use L-BFGS-B. Perhaps this should be taken farther by enabling y2eq to output any instance of a functional form without the use of L-BFGS-B. That is, update the dataset to have the target be an instance of a functional form rather than a functional form and allow the additional tokens necessary (`.` and `0`).

6.2.3 The cost challenge

In this work, we have shown that the way in which symbolic cost is calculated in regard to the `PAD` token results in a bias toward longer equations. Perhaps there is something about functional forms compared with natural language that makes this problem more extreme. Are their as many sentences in natural language dataset as their are in y2eq datasets that share the same beginning? There are certainly examples of sentences that start the same; for instance: "I want to eat cake.", "I want to eat cake in the living room.", and "I want to each cake in the living room with my friends.". But, we are not sure if this is as common as it is in the y2eq dataset.

Perhaps the result of the padding experiment can be improved by normalizing symbolic cost based on target tokens. Since the `PAD` token makes up about half the target tokens in the dataset, including `PAD` tokens in symbolic cost may put too much focus on the `PAD` tokens which in many cases does not matter much. Instead of including `PAD` tokens in the calculation of sym-

bolic cost, a penalty term could be used to discourage predicted and target functional forms of different lengths.

The challenge with the cost functions (used to train the model) is mainly the choice between numeric cost and symbolic cost. The experiments so far have only used symbolic cost. As many other symbolic regression methods use numeric cost, it seems likely that y2eq would benefit from the use of numeric cost during training.

In future work, we would like to train eq2y (see Section 4.3.3) to predict y-values from an equation in order to use this model to train y2eq with numeric cost (in addition to symbolic cost). The combination of y2eq and eq2y is called y2eq2y (see Section 4.3.3) and acts somewhat like an autoencoder. It is important to train a neural network to do the job of eq2y because it is unclear how to differentiate the process otherwise.

Alternatively, evolutionary methods could be used to train y2eq on numeric and symbolic costs. The down side to this approach is that a population of neural networks must be evaluated to determine the next weight update, which implies longer training time.

References

[1] Aftab Anjum, Fengyang Sun, Lin Wang, and Jeff Orchard. A novel neural network-based symbolic regression method: Neuro-encoded expression programming. In *Lecture Notes in Computer Science*, pages 373–386. Springer International Publishing, 2019.

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[3] Luca Biggio, Tommaso Bendinelli, Aurelien Lucchi, and Giambattista Parascandolo. A seq2seq approach to symbolic regression.

[4] Qi Chen, Bing Xue, and Mengjie Zhang. Differential evolution for instance based transfer learning in genetic programming for symbolic regression. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion on - GECCO '19*, pages 161–162. ACM Press, 2019.

[5] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[6] Christopher Choy, Junha Lee, René Ranftl, Jaesik Park, and Vladlen Koltun. High-dimensional convolutional networks for geometric pattern recognition. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11227–11236, 2020.

[7] Miles Cranmer, Alvaro Sanchez-Gonzalez, Peter Battaglia, Rui Xu, Kyle Cranmer, David Spergel, and Shirley Ho. Discovering symbolic models from deep learning with inductive biases. *arXiv preprint arXiv:2006.11287*, 2020.

[8] Thi Thu Huong Dinh, Thi Huong Chu, and Quang Uy Nguyen. Transfer learning in genetic programming. In *2015 IEEE Congress on Evolutionary Computation (CEC)*, pages 1145–1151. IEEE, IEEE, May 2015.

[9] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[10] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N Dauphin. Convolutional sequence to sequence learning. In *International Conference on Machine Learning*, pages 1243–1252. PMLR, 2017.

[11] Roger Guimerà, Ignasi Reichardt, Antoni Aguilar-Mogas, Francesco A Massucci, Manuel Miranda, Jordi Pallarès, and Marta Sales-Pardo. A bayesian machine scientist to aid in the solution of challenging scientific problems. *Science advances*, 6(5):eaav6971, 2020.

[12] Edward Haslam, Bing Xue, and Mengjie Zhang. Further investigation on genetic programming with transfer learning for symbolic regression. In *2016 IEEE Congress on Evolutionary Computation (CEC)*, pages 3598–3605. IEEE, IEEE, July 2016.

[13] Raban Iten, Tony Metger, Henrik Wilming, LĂdia del Rio, and Renato Renner. Discovering physical concepts with neural networks. *Phys. Rev. Lett.*, 124(1):010508, January 2020.

[14] Maarten Keijzer. Improving symbolic regression with interval arithmetic and linear scaling. In *European Conference on Genetic Programming*, pages 70–82. Springer, 2003.

[15] Samuel Kim, Peter Y Lu, Srijon Mukherjee, Michael Gilbert, Li Jing, Vladimir Čeperić, and Marin Soljačić. Integration of neural network-based symbolic regression in deep learning for scientific discovery. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[16] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[17] John R Koza and John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[18] Guillaume Lample and François Charton. Deep learning for symbolic mathematics. *arXiv preprint arXiv:1912.01412*, 2019.

[19] Dennis Lee, Christian Szegedy, Markus N Rabe, Sarah M Loos, and Kshitij Bansal. Mathematical reasoning in latent space. *arXiv preprint arXiv:1909.11851*, 2019.

[20] Li Li, Minjie Fan, Rishabh Singh, and Patrick Riley. Neural-guided symbolic regression with semantic prior. *arXiv preprint arXiv:1901.07714*, 2019.

[21] Mingsheng Long, Han Zhu, Jianmin Wang, and Michael I Jordan. Deep transfer learning with joint adaptation networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2208–2217. JMLR. org, 2017.

[22] Jiasen Lu, Jianwei Yang, Dhruv Batra, and Devi Parikh. Hierarchical question-image co-attention for visual question answering. *Advances in neural information processing systems*, 29:289–297, 2016.

[23] Trent McConaghy. Ffx: Fast, scalable, deterministic symbolic regression technology. In *Genetic Programming Theory and Practice IX*, pages 235–260. Springer, 2011.

[24] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017.

[25] Brandon Muller, Harith Al-Sahaf, Bing Xue, and Mengjie Zhang. Transfer learning. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion on - GECCO '19*, pages 350–351. ACM Press, 2019.

[26] Ji Ni, Russ H Drieberg, and Peter I Rockett. The use of an analytic quotient operator in genetic programming. *IEEE Transactions on Evolutionary Computation*, 17(1):146–152, 2012.

[27] Damien O'Neill, Harith Al-Sahaf, Bing Xue, and Mengjie Zhang. Common subtrees in related problems: A novel transfer learning approach for genetic programming. In *2017 IEEE Congress on Evolutionary Computation (CEC)*, pages 1287–1294. IEEE, IEEE, June 2017.

[28] Brenden K Petersen. Deep symbolic regression: Recovering mathematical expressions from data via policy gradients. *arXiv preprint arXiv:1912.04871*, 2019.

[29] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation.

In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 652–660, 2017.

[30] Subham Sahoo, Christoph Lampert, and Georg Martius. Learning equations for extrapolation and control. In *International Conference on Machine Learning*, pages 4442–4450. PMLR, 2018.

[31] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[32] Chuanqi Tan, Fuchun Sun, Tao Kong, Wenchang Zhang, Chao Yang, and Chunfang Liu. A survey on deep transfer learning. In *International conference on artificial neural networks*, pages 270–279. Springer, 2018.

[33] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

[34] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[35] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International conference on machine learning*, pages 2048–2057. PMLR, 2015.

[36] Ciyou Zhu, Richard H. Byrd, Peihuang Lu, and Jorge Nocedal. Algorithm 778: L-bfgs-b: Fortran subroutines for large-scale bound-constrained optimization. *ACM Trans. Math. Softw.*, 23(4):550â560, December 1997.